Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages

Hendrik Bünder
itemis AG, Bonn, Germany

Keywords: Textual Domain-Specific Languages, Model-Driven Development, Language Server Protocol, Case Study.

Abstract: Model-Driven Software Development using Domain-Specific Languages (DSL) has been widely adopted throughout research and industry. The language workbenches required to efficiently build Domain-Specific Languages and the associated editor support are often deeply integrated into a specific Integrated Development Environment (IDE). Thereby, the chosen Domain-Specific Language workbench predicts the IDE required to use the DSL. Yet, this IDE might not be the best choice for further implementing, testing, and debugging the generated code. A case study was conducted to analyze how the Language Server Protocol could be utilized to decouple the DSL implementation from a specific editor integrated into an IDE. First, the Language Server Protocol capabilities are exemplified by building editor support for an Entity-DSL that is integrated into two different IDEs. Second, a SWOT analysis is carried out to identify strengths and weaknesses as well as opportunities and threats for Domain-Specific Languages utilizing the Language Server Protocol. The case study's results indicate that the Language Server Protocol enables efficient multi-editor integration. Further, the results of the SWOT analysis imply potential benefits for cross-functional teams specifying a shared domain model.

1 INTRODUCTION

Model-Driven software development focuses on the abstract, formal description of domain models, that declaratively describe real-world concepts. Model-to-model and model-to-text transformations interpret the domain models and eventually create source code and configuration files that constitute the executable software system (Sendall and Kozaczynski, 2003). Textual Domain-Specific Languages are used to specify domain models using a textual concrete syntax. In contrast to general purpose programming languages such as Java, textual Domain-Specific Languages are tailored to a narrow problem area for which they offer concise and semantically rich notations (Hudak, 1997). Due to sophisticated editors that offer language smarts, such as code completion, goto definition, and validations, modelers create domain models efficiently and correctly.

In order to implement a Domain-Specific Language with an associated editor, language workbenches, such as MPS (Campagne, 2016), MontiCore (Krahn et al., 2014), or Spoofax (Kats and Visser, 2010) can be utilized. Yet, the mentioned language workbenches are deeply integrated into specific Integrated Development Environments, such as IntelliJ (JetBrains, 2018) or Eclipse (Vogel and Beaton, 2013). While the deep integration enables semi-automatic, efficient creation of language editors, it also causes a tool lock-in by interweaving the DSL editor with a specific IDE (Völter, 2013).

Yet, the IDE used for implementing, testing and debugging a software system should not be determined by the DSLs language workbench. Although most IDEs have support for the most popular programming languages, they usually have very mature support for a few specific programming languages. For example, the Eclipse IDE offers sophisticated editor support for Java (Arnold et al., 2005) and C++ (Stroustrup, 2000), while Visual Studio Code (Microsoft, 2018) has a focus on JavaScript (Mikkonen and Taivalsaari, 2007) and TypeScript (Bierman et al., 2014) editors. Further, the IDE integrated editor as well as testing and debugging support is differently mature. While Eclipse and IntelliJ both provide IDE features for Android development, IntelliJ offers a more stable and feature rich editor. Therefore, predefining the IDE by the DSLs language workbench can be disadvantageous for toolsmiths and tool users.

The latter are either forced to use two separate but specialized IDEs or one consolidated but not ideal IDE. Both approaches decrease developer productiv-

Bünder, H.
Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages.
DOI: 10.5220/0007556301290140
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 129-140
ISBN: 978-989-758-358-2

ity. The first by requiring a constant tool switch that causes slower and more error-prone responses (Monsell, 2003). The second, by not using the best tool available for tasks like testing and debugging.

For toolsmiths there is often no economically reasonable alternative to focusing on one IDE, because supporting multiple editors causes tremendous effort. Editor integration as well as computation of features, such as code completion or validations, have to be implemented for each IDE. Therefore, toolsmiths often decide to provide support for only one editor or IDE.

While the use of IDEs as primary environment comforts people with IT background, the large number of buttons, menus, and views often confuses non-technical users. Therefore, implementing support for only one editor not only causes the problems mentioned above but also excludes a group of potential users. All available language workbenches are focused on creating mature tooling for arbitrary DSLs, however, only for one specific IDE to integrate with.

The Language Server Protocol (LSP) which was intentionally built to separate the language smarts of general purpose programming languages from the editor integration, was recently adopted by the Xtext language workbench. Thereby, textual DSLs based on Xtext should be able to utilize all features of the Language Server Protocol. A case study was conducted to build an Entity-DSL that was integrated into two different IDEs. Language smarts and the two editor integrations are based on the Language Server Protocol. In addition, the case study includes a SWOT analysis carried out to identify the impact of the LSP on textual Domain-Specific Languages in general.

After analyzing related work Section 3 elaborates on architecture, features and limitations of the Language Server Protocol. In section 4 the Entity-DSL implemented in the context of the case study is described. Next, we outline the results of the SWOT analysis conducted to identify the potential impact of the LSP on textual domain specific languages. Finally, the results of the case study are discussed.

2 RELATED WORK

According to Tomassetti (Tomassetti, 2017a), an external Domain-Specific Language in contrast to an internal Domain-Specific Language has tool support. In order to build such tool support, there are different language workbenches available, that ease the creation of abstract syntax, parser, editor, and generators (Fowler, 2005). Language workbenches like MontiCore or MPS generatively create DSL tools to be integrated into Eclipse or IntelliJ, respectively. Xtext and Spoofax are two language workbenches that support Eclipse as well as IntelliJ integration. However, since both are primarily focused on Eclipse, the IntelliJ support is rather experimental for Spoofax (Kats and Visser, 2010) and lacking contributors for Xtext (Xtext, 2018).

In addition to language workbenches targeting specific IDEs, there is a variety of language workbenches that produce web-based editors. Popoola et. al. summarize such approaches as Modeling as a Service (MaaS) (Popoola et al., 2017). They identify two categories of MaaS, namely client-server and cloud-based. AToMPM, ModelBus, and DSLForge represent modeling platforms from the first category, that in some way require an installation on a local server. In contrast, GenMyModel, WebGME, CLOOCA, MORSE, and MDEForge require no installation and are accessible completely over the web. While all approaches enable browser-based editing of models, none of the approaches supports native integration into IDEs, such as Eclipse or IntelliJ.

The Language Server Protocol as introduced by Microsoft, RedHat, and Codeenvy in 2016 fills this gap (Microsoft, 2018). It aims at separating the computation of language editor features, such as code completion and validations, from the actual editor integration. Originally, invented to integrate different programming languages into different IDEs and editors, there is also an implementation for the Xtext language workbench. According to the list of Language Server implementations (Microsoft, 2018a), Xtext is by now the only language workbench supporting the LSP. While Rodriguez-Echeverria et. al. (Rodriguez-Echeverria et al., 2018) have already proposed a Language Server Protocol infrastructure for graphical modeling, little attention has been paid to the impact of the LSP on textual modeling.

3 LANGUAGE SERVER PROTOCOL

The number of programming languages is steadily increasing over the last decades. While new programming languages, such as Go (Go, 2018) or Swift (Apple, 2018), are gaining market shares (TIOBE, 2018), old programming languages, such as Cobol or Fortran, are staying as the basis of many legacy systems (Reuters, 2018). In addition, the number of integrated development environments is increasing as well. Since most IDEs support a variety of programming languages, IDE developers face the problem of keeping up with ongoing programming language evolution. At the same time language providers are inter-

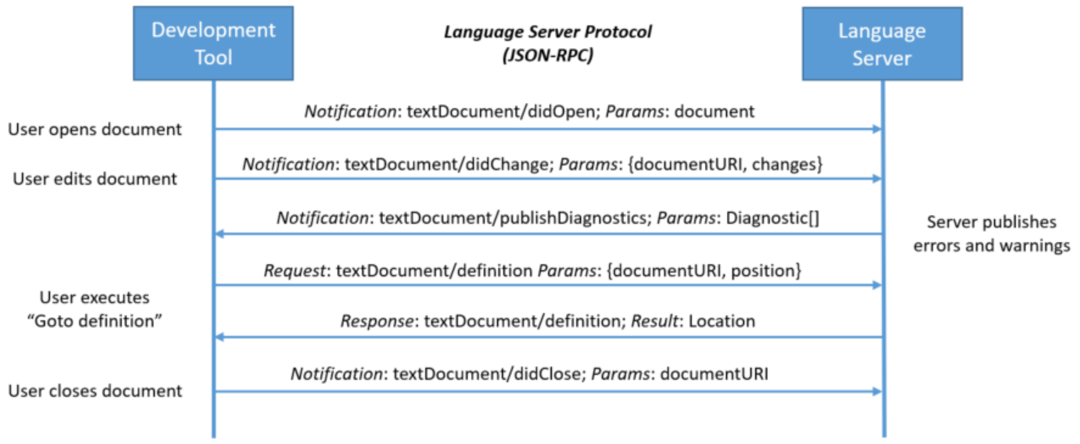

Development Tool
Language Server Protocol (JSON-RPC)
Language Server
User opens document
Notification: textDocument/didOpen; Params: document
User edits document
Notification: textDocument/didChange; Params: {documentURI, changes}
Notification: textDocument/publishDiagnostics; Params: Diagnostic[]
Server publishes errors and warnings
User executes "Goto definition"
Request: textDocument/definition Params: {documentURI, position}
Response: textDocument/definition; Result: Location
User closes document
Notification: textDocument/didClose; Params: documentURI

Figure 1: Interprocess communication during routine editing session (Microsoft, 2018).

ested in providing as many IDE integrations as possible to serve a broad audience. Consequently, integrating every language in every IDE leads to a *m-times-n* complexity.

The Language Server Protocol addresses this issue by separating the language-specific smarts from the integration into an editor. Features such as code completion, goto definition and refactoring are computed in a language-specific server process. The Language Server implementation itself can be implemented in any programming language. Yet, there are Software Development Kits (SDKs) encapsulating low-level communication as specified by the LSP for programming languages such as JavaScript or Java (Microsoft, 2018a). The server process communicates over a standardized protocol with a client process that integrates the language smarts into an IDE. The essential parts of the Language Server Protocol are the features specified by the protocol that client or server implement. By decoupling the language implementation from the editor integration the complexity for integrating all programming languages into all existing IDEs goes down to *m-plus-n* (Sourcegraph, 2018).

Since being introduced in 2016 more than 50 language servers for languages such as Java, TypeScript, and Cobol have been implemented. Moreover, there are editor integrations for more than 10 IDEs including widely adopted IntelliJ, Eclipse, and VSCode (Sourcegraph, 2018). Since textual DSLs are programming languages for a narrow set of problems (Völter, 2013), the LSP can be utilized to decouple language smarts from the editor for any DSL. However, by now Xtext is the only language workbench that includes LSP support.

3.1 Architecture of the Language Server Protocol

The Language Server Protocol is implemented by a server and a client process. The interprocess communication is defined by a protocol specifying the available capabilities (Microsoft, 2018). The protocol is based on the stateless and lightweight remote-procedure call protocol JSON-RPC. The transport agnostic protocol uses JSON as data format (JSON-RPC Working Group, 2018). Although JSON-RPC is designed for remote procedure calls over a network, server and client process not necessarily need to be on different physical machines. In fact, most Language Server implementations run two separate processes, but on the same physical machine (Tomassetti, 2017b).

Figure 1 shows a routine editing session. On the left-hand side, the Development Tool is shown and on the right-hand side, the Language Server implementation is presented. The communication that is represented by the arrows between the Development Tool and Language Server utilizes the Language Server Protocol for remote procedure calls. The editing session starts in the Development Tool where the user opens a document. The Development Tool notifies the Language Server that a document has been opened. Next, the user edits the document which is again forwarded to the server (textDocument/didChange). The server now analyses the changes within the document and sends detected errors and warnings back to the Development Tool.

The following use case starts with the user executing a "Goto definition", e.g. to jump to the declaration of a variable. The Development Tool sends a request to the Language Server giving the current cursor position within the document. Next, the position of the variable declaration is computed and returned to the Development Tool. The Development Tool integration is now in charge of opening the correct document and placing the cursor in the returned position. Finally, the user closes the document and the server is notified again. From here on the truth about the content of the files is persisted on the file system (Bumer et al., 2017).

The sample communication shows that the protocol is based on documents and positions within them. Thereby, the protocol itself is independent of a specific programming language. However, it is currently limited to text files. Consequently, binary file formats are not supported. In addition to programming languages, there is a variety of Language Server implementations for configuration languages, such as YAML (YAML, 2018) or XML (Microsoft, 2018a). Xtext is currently the only textual DSL workbench that supports the LSP. Therefore, languages implemented with Xtext can utilize the features of the Language Server Protocol .

3.2 Language Server Protocol Features

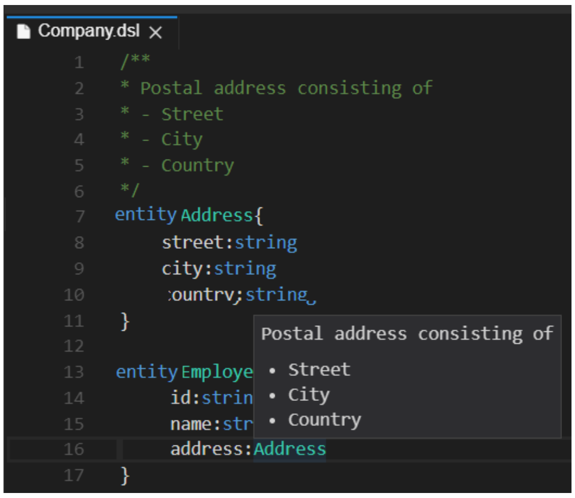
The Language Server Protocol currently in version 3.6 separates messages into header and content part. The first specifies content length and content type and is required for every message. The second contains the actual content of the message. Within the content of a message JSON-RPC 2.0 is used to describe requests, responses, and notifications. While request messages always require a response message, notifications do not (Microsoft, 2018b). In total the current LSP version defines more than 40 message types organized in five operational categories, namely general, window, client, workspace, and document (Rodriguez-Echeverria et al., 2018). The data format of the messages is JSON (JSON-RPC Working Group, 2018).

Since not every language as well as not every editor supports all features from the protocol, both use `Capabilities` to announce the features they support. During the initial request to the Language Server, the client provides its supported `Capabilities`. With the LSP 3.6 the `Capabilities` consist of `ClientCapabilities`, `TextdocumentClientCapabilities`, and `WorkspaceClientCapabilities`. The latter describe features the editor supports on the workspace level, such as operations to create, delete or rename resources. `TextdocumentCapabilities` contain the support for text hovers or type definitions. The `ClientCapabilities` group `TextdocumentCapabilities` and `WorkspaceClientCapabilities`. Additionally, they add `ExperimentalCapabilities` which are capabilities currently under development by a specific language or editor implementation.

In general, the Language Server ignores unknown `Capabilities` and interprets missing `Capabilities` as not being supported by the client. The Language Server itself answers to an initialization process by returning the `Capabilities` offered to provide language smarts (Microsoft, 2018c).

The focus of the protocol is on the language features implemented by the Language Server. While there is a large and steadily increasing number of language features included in the protocol, the community-driven LSP-site (Sourcegraph, 2018) cites “Code Completion”, “Hover”, “Goto Definitions”, “Workspace Symbols”, “Find References”, and “Diagnostics” as key features of a Language Server implementation.

Code Completion. The code completion request is sent to the server including the current document and the cursor position within the document. The Language Server computes completion items at the given position and returns them to the Development Tool. The client process is in charge of forwarding the completion proposals to the respective context menu within the editor from which the user can select the appropriate element. In case the computation of completions is expensive, the completion can be represented by a `handler`. As soon as the user selects the completion from the context menu, the `handler` executes another request to the Language Server to compute the full completion text. From the Language Server implementations listed at (Sourcegraph, 2018) nearly 90 percent support the code completion feature.



```
/**
 * Postal address consisting of
 * - Street
 * - City
 * - Country
 */
entity Address{
    street:string
    city:string
    :ountrv;string
}

entity Employe
    id:strin
    name:str
    address:Address
}
```

```
Postal address consisting of
• Street
• City
• Country
```

Figure 2: Theia Editor Integration - Hover Feature.

Hover. By sending a hover request to the server additional information to be shown in a hover menu is computed. Figure 2 illustrates the default hover menu integrated into the Theia IDE. The hover menu opened in line 16 contains the documentation of the `Address` entity that is referenced as type of the attribute `address` within the `Employee` entity. Additionally, text formatting including line-breaks, lists or indentations is supported by the hover menu feature. Around 88 percent of the listed Language Server implementations provide a hover capability.

Goto Definitions. The Goto Definition request returns the position within a document where the given symbol is defined. Since LSP version 3.6.0 "Goto Type Request" and "Goto Implementation Request" have been added. While the first returns the position where a given type is specified, the second returns the implementation location of a given symbol. The Goto Definition feature is supported by 83 percent of the listed Language Server implementations.

Workspace Symbols. The Workspace Symbol request takes a query string and reveals all matching symbols within the workspace. Out of the 58 Language Server implementations only 35 implement the workspace symbol request.

Find-References. Based on the current cursor position within the document, the find-references request returns a project-wide list of locations. Each of the locations references the symbol at the current cursor position. The find-references request is implemented by nearly 63 percent of the listed Language Server implementations.

Diagnostics. The Language Server is responsible for handling diagnostics. Depending on the kind of project, diagnostics might be handled per file or for a complete project. After the diagnostics, such as errors and warnings, are computed, the server sends them to the client. The client always replaces all diagnostics so that no merge on the client side is required. Therefore, a diagnostics notification with an empty list of diagnostics is interpreted as no errors or warnings occurred. 78 percent of the Language Server implementations support the diagnostics feature.

Besides the capabilities mentioned above, there are more advanced features such as code lens or signature help included in the Language Server Protocol (Microsoft, 2018b).

3.3 Limitations

The LSP provides a protocol for inter-process communication between a language-specific server and an editor specific client process. While the lightweight protocol enables the separation of concerns and reduces the integration complexity, it also has some inherent limitations.

First, the LSP currently assumes that client and server process have access to a shared file system. Consequently, client and server run on the same physical machine. Not only is the naming of server and client misleading, sharing a physical machine hinders scalability and fail-over mechanisms of the stateless server process. However, there are extensions to the Language Server Protocol, such as the Files Extension that enables the Language Server to work without a shared file system (Sourcegraph, 2016).

Second, the Language Server currently assumes one Language Server serves exactly one Development Tool (Microsoft, 2018b). In combination with running on a single physical machine, editing many different types of files at the same time, e.g. a Xtext DSL, a Java program, and a XML configuration file leads to three language servers running on a single computer. Further, every Language Server is built as a stand-alone application that does not communicate with other language servers. In contrast to native integrations, common tasks, such as parsing XML files, have to be implemented within each Language Server.

Thirdly, the Language Server Protocol currently focuses on editing the text documents of a programming, configuration or Domain-Specific Language. Yet, there is no inherent support for running, testing or debugging the specified programs. Hence, the editor might benefit from the decoupling achieved by applying the Language Server Protocol , but features mentioned above are still IDE-specific implementations. However, for the purpose of debugging the Debugging Server Protocol has been announced, that provides debugging support to be integrated into different editors(Kichwas Coders, 2018). Thereby, enabling IDE independent debugging support through a standardized protocol that is similar to the LSP.

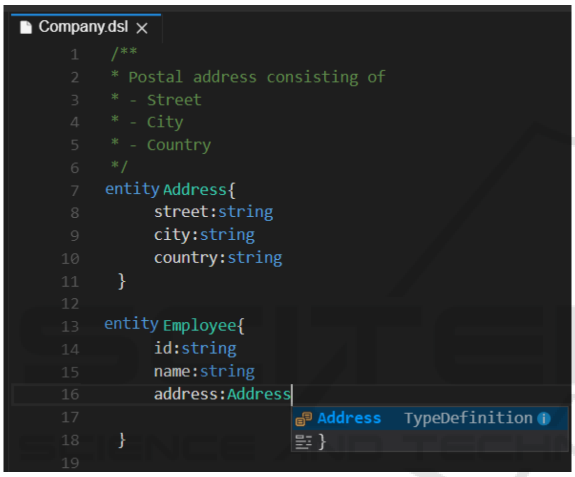
Fourthly, the Language Server Protocol has some shortcomings compared to native IDE integrations. Comparing the features available when editing a DSL natively integrated into the Eclipse IDE, there is sophisticated support for type hierarchies or project- and file-creation wizard (Vogel and Milinkovich, 2015). The gain of easy integration into multiple editors comes at the cost of losing some advanced features available in native integrations.

4 BUILDING AN ENTITY-DSL WITH THE LSP

In order to exemplify the capabilities of the Language Server Protocol a case study was conducted. An

Entity-DSL leveraging the LSP was implemented to investigate the necessary tasks to integrate with different IDEs. The Entity-DSL Language Server supports the key language features as stated by the community-driven LSP-site (Sourcegraph, 2018). To exemplify the integration in different editors, a Language Server client extension for Theia and Eclipse was implemented.

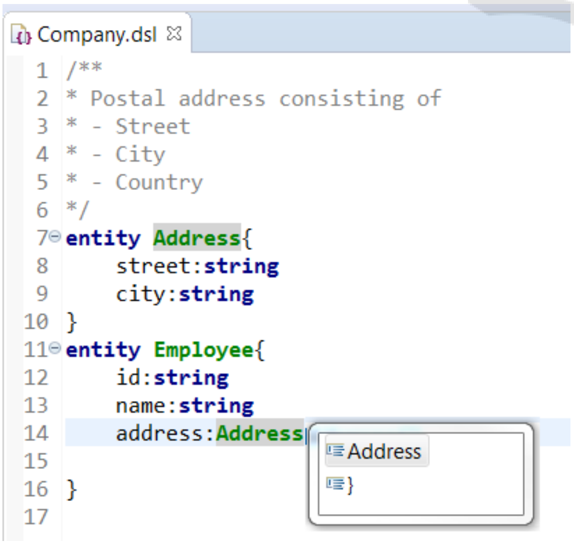
Figure 3 shows the editor within the Theia IDE, and Figure 4 displays the same language integrated into the Eclipse IDE. In both cases the same "Company.dsl" file is edited. While the completion proposal shown by both figures is computed by the Language Server process, the keyword highlighting was implemented specifically for each editor integration.



```
/**
* Postal address consisting of
* - Street
* - City
* - Country
*/
entity Address{
      street:string
      city:string
      country:string
 }

entity Employee{
      id:string
      name:string
      address:Address
                        Address    TypeDefinition
                        }
 }
```

Figure 3: Theia Editor Integration.



```
/**
* Postal address consisting of
* - Street
* - City
* - Country
*/
entity Address{
    street:string
    city:string
}
entity Employee{
    id:string
    name:string
    address:Address
                        Address
                        }

}
```

Figure 4: Eclipse Editor Integration.

The following subsections elaborate on implementing the Language Server parts to compute Entity-DSL smarts, as well as the client integration for Theia and Eclipse.

4.1 The Entity-DSL Language Server Implementation

Listing 1 shows an Xtext grammar in an EBNF-like format defining the Entity-DSL including entities with properties and operations. The Entity-DSL is a simplification of a typical textual DSL used in real world projects to specify data models and operations. The root grammar element is the `Model` that can hold an arbitrary number of `Entity` objects. Each `Entity` can extend another `Entity` and contains an unordered list of `Property` and `Operation` objects. While a `Property` has a `name` and a `type` attribute, an `Operation` consists of a `name`, an optional list of `Parameters` and a `returntype`. `Types` can either reference an `Entity` or a `Primitive` type, such as `string`, `number`, or `boolean`.

```
Model:
 entities+=Entity*;
Entity:
 'entity' name=ID ('extends' superType=Entity)?
  '{'
    features+=Feature*
  '}';
Feature:
 Property | Operation;
Property:
 name=ID ':' type=Type;
Operation:
 'op' name=ID
 '('(params+=parameter(','params+=parameter)*)?')'
  (':'returntype=Type)?;
Parameter:
 name=ID type=[Type];
Type :
 {Primitive} name=Primitive |  EntityReference;
EntityReference :
 {EntityReference} entityDefinition=[Entity];
Primitive :
 'number' | 'string' | 'boolean';
```

Listing 1: Grammar Rules for the Entity-DSL.

Based on the EBNF-like grammar, the Xtext language workbench creates the required source code to parse and link text files according to the specified grammar. The generated language server classes use the Language Server Protocol for Java (LSP4J) framework (Eclipse, 2018c) to create messages in accordance with the Language Server Protocol. LSP4J encapsulates the low-level JSON communication and defines a Java API that offers all features of the Language Server Protocol (Miro Spnemann, 2018). Since the Xtext language workbench generates the required classes for the key language features, the default Language Server implementation needs no further customization. Yet, it supports the features explained in Section 3.2.

4.2 Building a Client-extension for the Theia Editor

Theia is a cloud and desktop IDE that integrates many different languages based on the Language Server Protocol. It separates the graphical user interface from the backend processes. While the frontend either runs in a browser or as a desktop application, the backend process is executed locally or remotely, e.g. on a cloud-based infrastructure (Theia.org, 2018).

In order to introduce editor support for the Entity-DSL in Theia a backend extension is required that configures the Language Server. Since Theia uses dependency injection (Fowler, 2004) the required backend extension can be injected as shown by Listing 2. The `EntityDslContribution` extends the `BaseLanguageServerContribution` from Theia's LSP-specific framework classes. As shown by Listing 2, the Language Server can either run as local sub-process or remotely using web-socket communication (Fette and Melnikov, 2011). The details for establishing the connections are encapsulated in Language Server client specific libraries.

```
@injectable()
class EntityDslContribution extends
     BaseLanguageServerContribution {

 start(clientConnection: IConnection): void {
  let socketPort = getPort();
  if (socketPort) {
   this.connectToRemoteServer(clientConnection,
        socketPort)
  } else {
   this.connectToLocalServer(clientConnection)
  }
 }
}
```

Listing 2: Registering a Language Server Extension in Theia.

By registering a backend extension, the Theia editor is ready to interpret the Entity DSL. Yet, the highlighting of keywords and comments is handled by Theia's frontend process. Therefore, an additional frontend extension is added to support syntax highlighting. In addition, the frontend extension specifies the file suffix for which the Language Server should be used. Listing 3 shows the part of the frontend extension where the "globPattern" method of the `EntityDslClientContribution` is overwritten to specify the file extension the Language Server is bound to.

```
@injectable()
 export class EntityDslClientContribution
   extends BaseLanguageClientContribution
    {
     protected get globPatterns() {
          return ['**/*.dsl'];
         }
}
```

Listing 3: Binding the ".dsl" file extension to the Language Server.

Based on the backend and frontend extension the Theia IDE can interpret Entity-DSL models as shown by Figure 3.

4.3 Building a Client-extension for the Eclipse IDE

The Xtext language workbench makes heavy use of the Eclipse IDE for providing sophisticated, natively integrated editor support for its DSLs. However, instead of utilizing this native integration it will be shown how the Entity-DSL can be integrated into the Eclipse IDE using the Language Server for Eclipse (LSP4E) framework (Eclipse, 2018b).

The integrated development environment Eclipse has an extensible plugin architecture (Vogel and Beaton, 2013). The IDE capabilities are extended by providing Eclipse plugins for specific purposes. Consequently, to enable LSP-based support for the Entity-DSL an Eclipse-Plugin is provided. The Entity-DSL plugin extends the LSP4E Plugin using well-defined extension points (Gamma and Beck, 2004). The LSP4E plugin offers the DSL-agnostic "org.eclipse.lsp4e.languageServer" extension point. As shown by Listing 4, the Entity-DSL plugins provides an extension in the class `EntityDslLanguageServerClass`. While the LSP4E plugin ensures that the Entity-DSL extension is called at the correct time, the Entity-DSL plugin handles the language-specific implementation.

```
<extension
 point="org.eclipse.lsp4e.languageServer">
 <server
  id="org.eclipse.lsp4e.languages.dsl"
  class="org.eclipse.lsp4e.languages.dsl.
      EntityDslLanguageServer"
  label="Entity-DSL Language Server">
 </server></extension>
```

Listing 4: Registering a Language Server Extension in Eclipse.

Listing 5 shows the implementation of the `EntityDslLanguageServer` that extends the LSP4E framework class `ProcessStreamConnectionProvider`. As soon as an editor for the ".dsl" file is opened a Language Server is started as sub-process. The `createLauncherCommand` method returns a command to start a Language Server process as sub-process. Afterwards, the working directory for the Language Server is computed.

```
public class EntityDslLanguageServer extends
     ProcessStreamConnectionProvider {
 public MyDslLanguageServer() {
  setCommands(createLauncherCommand());
  setWorkingDirectory(workingDirectory());
 }
}
```

Listing 5: Entity-DSL Extension.

In addition, there is a `ProcessOverSocketStreamConnectionProvider` class in the LSP4E plugin that can be extended in case the connection to the server should be handled by web-socket communication. Thereby, the connection could be established to an already running Language Server.

```
public class EntityPresentationReconciler extends
     PresentationReconciler {

private Set<String> keywords = new HashSet<>(
     Arrays.asList(new String[] {
"entity", "op", "string", "number","boolean","op"
}));
}
```

Listing 6: Keyword Definition for Syntax Highlighting.

To ensure correct highlighting for the Eclipse IDE editor the `PresentationReconciler` needs to be extended. The `EntityPresentationReconciler` specifies syntax highlighting for keywords and comments of the Entity-DSL. Listing 6 shows the definition of all keywords of the Entity-DSL to be highlighted.

Introducing Language-Server-based support for the Entity-DSL in Theia and Eclipse is similar on a conceptual level. Both integrations require a specific implementation to support syntax highlighting. Further, both extensions have to take care of starting the Language Server. Yet, the two integrations have a lot of differences in the details, such as architecture, programming language, and API.

4.4 Results for Implementing the Entity-DSL

In order to indicate the effort required for creating a DSL with LSP support, we measured the time required to implement and integrate the Entity-DSL. Building and integrating the Entity-DSL into Theia and Eclipse can be divided into three major tasks. First, the Language Server Implementation was done using the Xtext language workbench. Since the Language Server was generated based on the Xtext grammar file, the first runnable version was done in about two hours.

Table 1: Effort for implementing the Entity-DSL.

Task	**Time (in minutes)**
Language Server Implementation	127
Theia Editor Integration	414
Eclipse Editor Integration	317

Second, the Theia integration was implemented requiring more manual implementation. Since frontend and backend extension had to be implemented, the overall effort for a first running version of the editor integration took about 7 hours. Thirdly, the Eclipse integration was implemented in a little bit more than 5 hours, due to the experience gained when writing the Theia extension.

For all three parts the implementation was only possible in such a short amount of time, because there are SDKs available that encapsulate low-level communication including marshaling and demarshaling.

5 IMPACT ON TEXTUAL DOMAIN-SPECIFIC LANGUAGES

In addition to implementing the Entity-DSL a SWOT analysis was carried out as part of the case study to get a holistic view on the impact of the Language Server Protocol on textual Domain-Specific Languages. The analysis of strengths, weaknesses, opportunities, and threats is used in a business context regularly for strategic planning (Helms and Nixon, 2010). After analyzing the internal strengths and weaknesses of the Language Server Protocol the external opportunities and threats are examined. The analysis reveals the strengths that benefit the spread of the Language Server Protocol and thereby the impact on textual DSLs. Additionally, it identifies weaknesses and threats that need to be removed or mitigated, respectively.

5.1 SWOT Analysis

Strengths. The Language Server Protocol provides a standardized protocol for inter-process communication between a Development Tool and a Language Server. Thereby, the language smarts required to compute editing operations for source code, configuration files, and textual Domain-Specific Languages only have to be implemented once. Consequently, support for new programming languages or textual DSLs can be introduced into different IDEs and editors quickly.

Since the protocol is based on the JSON-RPC standard, the data is represented by the lightweight JavaScript Object Notation format. Thereby, the payload between server and client is reduced to a minimum. Although LSP is currently used mainly with two processes running on the same physical machine, the efficient data exchange format forms an important prerequisite for communicating over a network. Additionally, the delays due to data exchange are minimized leading to better user experience when editing text documents.

While the Language Server Protocol specifies a set of language capabilities that a client or a server can implement, it also allows for custom capabilities. Using custom capabilities enables toolsmiths to implement language or editor specific features. Yet, the editor integration has to be built manually. Thereby, custom protocol features might not be supported by all editors or languages. Nevertheless, language extensions that appear to be reasonable for the majority of programming languages may be added to the protocol.

Especially for DSLs, the tool users are not always developers. Instead, team members with a business background want to engage in creating domain-specific models, e.g., to specify insurance contracts or banking products (Völter, 2013). The ability to integrate LSP-based editors seamlessly into browser-based applications lowers the barrier for non-technical experts. By accessing and editing the DSL through a browser-based interface, installation and distribution efforts are minimized. At the same time, web editors may not be as confusing as integrated development environments from a non-technical user's perspective.

Weaknesses. The LSP is implemented under the assumption that there is one Language Server per tool. While this is appropriate for a single language approach it has some shortcomings regarding multi-language approaches and cross-language referencing. Since a Language Server is started for every kind of language on the same physical machine (s. Section 3.3) as the editor, the choice of a programming language to implement the Language Server becomes significant. If the Language Server is implemented in Java, then a Java Virtual Machine (JVM) is started for every language. Assuming an average memory consumption of 1 GB per JVM and editing a DSL, a Java program, and a XML file at a time, 3 GB memory is allocated. Depending on the size and complexity of the projects and the computers used, this can result in poor user experience. However, since the programming language used to implement the Language Server is independent of the programming language of the Development Tool, toolsmith are free to use languages with a smaller memory footprint, such as JavaScript.

Additionally, language servers are currently implemented to be independent. While this enables separated life-cycles of servers, it also forces each language implementation to re-implement potentially reusable concepts of other languages. If a DSL, for example, references Java classes on the classpath, it has to implement the access to the classpath and the classes on it. In contrast, a native IDE integration, e.g., in Eclipse, allows the DSL implementation to reuse functionality from other plugins, such as the Java Development Tools (JDT) plugin (Eclipse, 2018a).

As Section 4 has shown, each editor integration requires its own specific extension. Therefore, the language provider still needs to specify which editors to support. Consequently, the knowledge about how to provide extensions for Eclipse, Theia, etc. is still required. Further, since every integration is a little bit different, they all have to be tested to guarantee the same user experience, e.g., keywords being highlighted or code completion proposals being sorted correctly.

The Language Server is currently limited to support programming languages and thereby textual DSLs that are programming languages for a specific problem domain. However, frameworks like IntelliJ's Meta Programming System (MPS) offer a powerful language workbench for implementing editors for projectional editing of DSLs (Fowler, 2005). In addition, graphical notations are often beneficial to present a quick overview of complex relations. Both are not supported by the Language Server Protocol. Thereby, excluding a large number of powerful Domain-Specific Language implementations.

Opportunities. Domain-specific languages are widely adopted through research and industry. Mature language workbenches such as Xtext enable fast creation of Domain-Specific Languages. However, the DSL usage is often bound to one specific IDE. At the same time, software developers demand more freedom when choosing their IDE. In multi-layered software architectures a variety of programming languages are utilized that are edited best with different IDEs. For example, the Javascript code for the frontend might be edited best with the Theia IDE, while the Java backend code is edited best with the Eclipse IDE. Yet, a holistic DSL aims to describe parts of both architectural layers, e.g., to provide consistency between backend and frontend.

The Language Server Protocol solves the problem of writing specific extensions or plugins for every potential editor and thereby enables a fast spread of the DSL. Given an industry that constantly adds new DSLs and editors, the Language Server Protocol creates the opportunity to provide a Domain-Specific Language in an environment that the user is most familiar with.

By introducing abstract Domain-Specific Languages, non-technical team members are enabled to engage in the process of describing a software system on a technology agnostic level. Yet, DSLs integrated into heavyweight IDEs cause a lot of distraction. The graphical user interface is often perceived by non-technical users to be overloaded with menus, buttons, and views. Further, additional software has to be installed and maintained to use the DSL. Besides being integrated into client-based IDEs a Language Server client extension can also be built for web-based editors such as Monaco (Microsoft, 2018). By editing DSLs through a browser-based editor that does not require any additional installations or new tools to learn, the entry barrier for non-technical users is lowered. Consequently, each team member can work on a shared ubiquitous domain model from his or her preferred editor environment. Further, the development activities of non-technical and technical team members are closely integrated. They not only use the same Domain-Specific Language to specify a software system, but they are also allowed to use the editor of their choice. Thereby, the domain-model can be created through two differently skilled stakeholders from their favorite editing environment.

Threats. While there is currently no comparable alternative to decoupling the language smarts from the editor integration, the threats do not lie in other technologies but rather in semantically rich programming languages. Further, multi-notational DSLs that integrate forms, tables, graphs and text pose a real threat.

New programming languages, such as Scala, D, or Go are more expressive than programming languages such as Java or C. Additionally, frameworks provide high-level APIs that ease the development of complex processes. More expressive languages and framework diminish the need for low-level textual DSLs, e.g. to describe entities or service interfaces, since the advantage gained from transformation and generation is small. Consequently, Domain-Specific Languages must provide higher levels of abstraction to create benefits.

In addition, multi-notation support, e.g. for tables, charts or formulas becomes more important. Although, projectional DSLs built with language workbenches such as MPS are bound to a specific IDE, they are a valuable alternative to parser-based DSLs. If the Language Server Protocol is not enhanced to also handle projectional editor based languages, a growing market share of DSLs is out of reach.

While current projectional DSL workbenches do not offer web editor support, there are browser-based multi-notation language workbenches. WebGME (Maróti et al., 2014), for example, enables the creation of Domain-Specific Languages through a web-client. Especially, for high-level, graphical notations WebGME is an alternative to parser-based languages built with the LSP.

6 DISCUSSION

The case study has shown that Entity-DSL language and editor can be separated easily by using the Language Server Protocol. At the same time, the integration into multiple different Development Tools is possible. Section 4.4 indicates that the creation of the Language Server itself is relatively fast, due to the high ratio of generated source code. Although each editor integration has its own programming language and API, basic concepts are the same. Therefore, the implementation of the second editor integration into Eclipse was slightly faster than the Theia integration. Further, the SWOT analysis revealed that the LSP is concerned with the right problems in an industry where the number of DSLs and Development Tools is increasing steadily. However, by focusing on textual DSLs valid and powerful alternatives such as graphical or projectional DSLs are not covered. Nevertheless, since the Xtext language workbench supports the LSP, the number of DSLs utilizing the protocol will increase in the future.

The Language Server Protocol has already had an impact on Domain-Specific Languages. First, the widely used Xtext language workbench has already adopted the LSP. Further, the language workbench automatically creates support for the main language features as listed by the community-driven LSP-site (Sourcegraph, 2018). Thereby, the Language Server Protocol has defined the capabilities a language must have to comply with the minimal editor-support "standard". Second, the LSP contains features that are currently not supported by the Eclipse native integration, e.g., the signature help feature. Since this capability can be included in DSLs built with the Xtext workbench, the LSP has introduced new features. Thirdly, DSLs built with the Language Server Protocol will in the future be edited by a broader group of potential users, such as developers, business analysts, and testers due to the relatively easy integration into multiple editors. Thereby, the LSP will foster the collaboration in cross-functional teams.

By enabling multi-editor integration, LSP-based textual DSLs become a valid alternative to web-based approaches. Domain-Specific Languages backed by the Language Server protocol have the advantage that they can be integrated into IDEs as well as into browsers. Thereby, enabling all team members to contribute to shared domain models from the editor that best suits their needs. The technological limitations, such as a shared physical machine and no direct communication between language servers, are outweighed by the advantages of decoupling language smarts and editor integration.

7 CONCLUSIONS

The paper has shown how the LSP can be leveraged to integrate one DSL in a variety of editors. Although the integration into every additional editor came at the cost of providing a new extension for the specific IDE, the overall integration costs are far below implementing an IDE-specific integration. Further, the SWOT analysis has shown that the LSP has the potential to provide textual DSLs to many different user groups and thereby foster collaboration. However, focusing solely on textual DSLs is also the largest disadvantage, since it leaves out projectional and graphical approaches.

The impact of the Language Server Protocol on textual Domain-specific languages is threefold. First, the six main features as listed by Section 3.2 can be seen as the minimum feature set that should be supported by any DSL editor. Second, the effort required to support different IDEs and editors for any given textual DSL is significantly reduced by the Language Server Protocol. Thirdly, by liberating the language smarts from the editor integration users can specify domain models without installing heavyweight IDE, e.g. through a browser-based editor.

The fact that each Language Server is built to serve one Development Tool and has no inherent connection to other Language Servers introduces some disadvantages. Additional research is required to analyze whether the separation leads to consequent reimplementation of basic features. Further, having one Language Server per Development Tool on the same physical machine should be investigated in industry size projects to estimate the effect on the overall performance.

The contribution of this paper based on the conducted case study is two-fold: First, we showed how to apply the LSP to implement an Entity-DSL and integrate its editor into the Theia and Eclipse IDE. Second, a SWOT analysis identified internal strength and weaknesses of the LSP and evaluated them against current external conditions in form of opportunities and threats. Based on the first-hand experiences from utilizing the LSP and the results of the SWOT analysis the potential impact on textual DSLs was analyzed.

REFERENCES

Apple (2018). About swift.

Arnold, K., Gosling, J., and Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

Bierman, G., Abadi, M., and Torgersen, M. (2014). Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer.

Bumer, D., Gamma, E., and McBrean, S. (2017). What is the language server protocol (lsp).

Campagne, F. (March 2016). *The MPS language workbench: Meta Programming System*. campagnelab, [New York, NY], third edition, version 1.5.1 edition.

Eclipse (2018a). Eclipse java development tools (jdt).

Eclipse (2018b). Eclipse lsp4e.

Eclipse (2018c). Eclipse lsp4j.

Fette, I. and Melnikov, A. (2011). The websocket protocol. Technical report.

Fowler, M. (2004). Inversion of control containers and the dependency injection pattern.

Fowler, M. (2005). Language workbenches: The killer-app for domain specific languages.

Gamma, E. and Beck, K. (2004). *Contributing to Eclipse: principles, patterns, and plug-ins*. Addison-Wesley Professional.

Go (2018). The go programming language.

Helms, M. M. and Nixon, J. (2010). Exploring swot analysis where are we now?: A review of academic research from the last decade. *Journal of Strategy and Management*, 3(3):215–251.

Hudak, P. (1997). Domain-specific languages. *Handbook of programming languages*, 3(39-60):21.

JetBrains (2018). Intellij idea.

JSON-RPC Working Group (2018). Json-rpc 2.0 specification.

Kats, L. C. and Visser, E. (2010). The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA. ACM.

Kichwas Coders (2018). Debugging protocol vs. language server protocol.

Krahn, H., Rumpe, B., and Völkel, S. (2014). Efficient editor generation for compositional dsls in eclipse. *arXiv preprint arXiv:1409.6625*.

Maróti, M., Kecskés, T., Kereskényi, R., Broll, B., Völgyesi, P., Jurácz, L., Levendovszky, T., and Lédeczi, Á. (2014). Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure. *MPM@ MoDELS*, 1237:41–60.

Microsoft (2018). Code editing. redefined.

Microsoft (2018a). Implementations - language servers.

Microsoft (2018b). Language server protocol specification - base protocol.

Microsoft (2018c). Language server protocol specification - initialize.

Microsoft (2018). Monaco editor - about.

Microsoft (2018). Overview - what is the language server protocol.

Mikkonen, T. and Taivalsaari, A. (2007). Using javascript as a real programming language.

Miro Spnemann (2018). The language server protocol in java.

Monsell, S. (2003). Task switching. *Trends in Cognitive Sciences*, 7(3):134 – 140.

Popoola, S., Carver, J., and Gray, J. (2017). Modeling as a service: A survey of existing tools. In *MODELS (Satellite Events)*, pages 360–367.

Reuters (2018). Cobol blues.

Rodriguez-Echeverria, R., Izquierdo, J. L. C., Wimmer, M., and Cabot, J. (2018). Towards a language server protocol infrastructure for graphical modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, pages 370–380, New York, NY, USA. ACM.

Sendall, S. and Kozaczynski, W. (2003). Model transformation: the heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45.

Sourcegraph (2016). Files extension to the lsp.

Sourcegraph (2018). Languageserver.org.

Stroustrup, B. (2000). *The C++ programming language*. Pearson Education India.

Theia.org (2018). Theia - cloud and desktop ide.

TIOBE (2018). Tiobe index for november 2018.

Tomassetti, F. (2017a). The complete guide to (external) domain-specific languages.

Tomassetti, G. (2017b). Why you should know the language server protocol.

Vogel, L. and Beaton, W. (2013). *Eclipse IDE: Java programming, debugging, unit testing, task management and Git version conrol with Eclipse*. Vogella series. Vogella, [Lexington, Ky], 3rd ed. edition.

Vogel, L. and Milinkovich, M. (2015). *Eclipse Rich Client Platform*. vogella series. Lars Vogel.

Völter, M. (2013). *DSL engineering: Designing, implementing and using domain-specific languages*. CreateSpace Independent Publishing Platform, Lexington, KY.

Xtext (2018). Idea support.

YAML (2018). Yaml ain't markup language.