On the Suitability of LSP and DAP for Domain-Specific Languages

Josselin Enet, Erwan Bousse, Massimo Tisi, Gerson Sunyé

On the Suitability of LSP and DAP for Domain-Specific Languages

Josselin Enet*, Erwan Bousse†, Massimo Tisi‡, Gerson Sunyé§

*†‡§Nantes Université, École Centrale Nantes, IMT Atlantique, CNRS, LS2N, UMR 6004, France
*Email: josselin.enet@univ-nantes.fr
†Email: erwan.bousse@univ-nantes.fr
‡Email: massimo.tisi@imt-atlantique.fr
§Email: gerson.sunye@univ-nantes.fr

***Abstract*—Domain-Specific Languages (DSLs) help manage the growing complexity of systems by facilitating their description and execution or simulation via tailored languages. A large part of the development costs of a DSL comes from building the associated tools it requires, such as an editor or a debugger. To reduce these costs, the Language Server Protocol (LSP) and Debug Adapter Protocol (DAP) enable the creation of generic tooling interfaces which rely on standardized services exposed by languages. However, as these protocols have been designed for General Purpose Languages (GPLs), their applicability to DSLs has no yet been extensively studied. In this paper, we analyze both LSP and DAP, with an emphasis regarding their relevance for the development of tooling for DSLs. We provide both a high-level insight into these protocols, such as a dependency graph of their services, and a more fine-grained qualitative analysis of each service. We show that while some services defined by these two protocols can be provided by any DSL, others make strong assumptions on the concepts that should be part of the considered DSL. Conversely, domain-specific concepts available in some DSLs are not exploitable through these protocols, thus reducing the capabilities of generic tools.**

***Index Terms*—Programming Languages and Software, Algorithm/protocol design and analysis, Testing and Debugging, Document and Text Editing.**

I. Introduction

Domain-Specific Languages (DSLs) are languages explicitly tailored for a given concern, providing abstractions and notations closely related to the concepts manipulated by domain experts. DSLs can serve to produce either programs or models[1], i.e. abstractions of real systems. Using DSLs has multiple benefits, such as enhanced productivity thanks to the reduction of implementation details, or clearer communication between the actors involved in the development [1]. In addition, DSLs often come with their own set of tools, further improving productivity and ease of use. Still, the ad-hoc implementation of specialized tooling has a high development cost.

Part of this cost comes from the effort of integrating language tooling in existing user interfaces (UIs). A possible solution to address this cost is protocol-based communication between the UI and an independent component responsible for language logic. This structure presents benefits both for toolmakers and language engineers:

- By using these protocols, a UI automatically supports all languages that provide the proper services, independently of each language implementation.
- By providing the correct services, a language can directly be supported by all UIs able to communicate through these protocols, allowing for a better adoption. Additionally, instead of each UI having its own implementation of the internal logic of the language, they all rely on the same component, improving overall consistency.

Two notable examples of such protocol-based approaches are the Language Server Protocol [2] (LSP), focused on textual editing, and the Debug Adapter Protocol [3] (DAP), tailored for debugging. LSP in particular had a considerable impact on the IDE ecosystem, with a wide range of compatible IDEs and language servers available[2]. However, these protocols are geared towards imperative, object-oriented General Purpose Languages (GPLs). As such, their usage for DSL tooling may be hindered by an intrinsic bias. Providing DSLs access to these protocols would greatly reduce the implementation effort of their associated tooling, but the suitability of LSP and DAP for DSLs has not yet been formally studied.

In this work, we propose a review of these two protocols, in which we focus on the following concerns: *1) providing a high-level, concise reading grid to improve the understanding of these protocols*; and *2) assessing the suitability of these protocols for the development of DSL tooling*. To achieve these goals, we provide a classification of the services contained in the protocols. This classification of services is based on multiple criteria, such as the language concepts they rely on, their relation to concrete syntax and their mandatoriness.

The remainder of this paper is structured as follows: we provide in Section II further background on domain-specific tooling and protocol-based architectures. Section III describes the process used in our review of LSP and DAP, and shows the results obtained. We use these results in Section IV to discuss the relevance of the considered protocols for the development of DSL tooling. Finally, we position this paper w.r.t. related

[1]Literature sometimes differentiates between Domain-Specific Languages (DSLs) and Domain-Specific Modeling Languages (DSMLs). We don't judge this distinction necessary in this paper: since all DSMLs are DSLs, findings related to DSLs presented in this work are also applicable to DSMLs.

[2]https://langserver.org/

work in Section V, and Section VI concludes by highlighting key results and bringing up future research possibilities.

II. Background

In this section, we further describe the activities for which LSP and DAP were created—i.e. textual editing and interactive debugging—and explore the specificities of such activities when applied to DSLs. We also provide an overview of the idea behind protocol-based approaches for language tooling.

A. Domain-Specific Tooling

To help programers with their time-consuming and error-prone activity, multiple techniques and helpers have been adopted over the years.

One aspect of programing impacted by such tooling is textual editing, which is the practice of writing programs through a textual concrete syntax. Operations provided by textual editing tooling is affected by the domain of each language. For GPLs, common operations can be identified and are usually provided by language tooling: these include auto-completion, go-to operations, and renaming facilities. In addition, languages manipulating other concepts may benefit from more advanced operations. For object-oriented GPLs, a typical specific operation is to display and / or manipulate the type hierarchy of a class. For a State Machine DSL, a specific "Remove State" operation would remove both the state declaration and all existing transitions referencing this state.

Another aspect is debugging, i.e. the process of understanding the execution of a program, often in an attempt to find and fix bugs. This process can be interactive [4], which means it takes place during the execution of the program itself. Multiple features are very commonly found in debuggers for GPLs. Control over the execution is usually handled through a *Pause* and *Continue* operator, as well as *stepping* operators. *Breakpoints* are conditions that can be apposed to a program, and pause the execution when verified. However, these usual operations may again not be sufficient for DSLs. Chiş et al. [5] implement domain-specific debugging operations for mutiple DSLs: a diff view is implemented for the testing language SUnit, comparing expected and actual results; a variety of stepping operators are provided for the parsing language PetitParser, for instance to step until a production is reached, or until an input matching failure is reached.

B. Protocol-Based Architecture

Language tooling has recently seen a surge in interest for protocol-based architectures, following the foundations laid by LSP. The main idea behind this approach is to separate language tooling in two distinct components: a reusable component responsible for handling language logic, and a UI through which end-users can trigger operations exposed by the first component. This separation presents important benefits in terms of reuse: languages and UIs alike only have to implement communication through a given protocol once. Additionaly, these protocols usually communicate through the network. This allows for UIs and languages existing in different technological spaces to interact with each other without further difficulties. For instance, the Eclipse IDE implemented in Java can easily communicate with the implementation of LSP for the Typescript language (itself written in Typescript).

LSP and DAP are notable examples of such protocols, but others have also emegerged: the Graphical Language Server Protocol [6] (GLSP) is geared towards editing for graphical languages; the Build Server Protocol [7] (BSP) handles the build process for different phases, such as compiling, execution or testing.

III. Protocols Review

This section presents the results of our evaluation of both LSP and DAP regarding our research questions. We begin by describing the review process followed to conduct this evaluation, then we present the results obtained for the two protocols.

A. Review Process

Fig. 1 presents the methodology followed for our analysis of LSP and DAP. The different steps depicted in this methodology are all based on manually inspecting the specification of each protocol. In these protocols, services are provided either by the UI or by the server (i.e. language server for LSP or debug adapter for DAP). We include both kinds of services in our review, which begins with four independent steps.

First, we identify the services that must always be provided (either by the UI or the server), and those that are completely optional. This classification is achieved by examining *capabilities*, a notion present in both protocols. Capabilities are exchanged during the initialization of the communication; the UI and the language can declare whether they implement some services or not. Capabilities are data structures tied to one or multiple services. Through the attributes present in a capability, a component can declare whether its associated services are implemented or not. We consider that a server-side service is mandatory if it has no associated capability that can be declared by the server. We consider that a UI-side service is mandatory if a mandatory server-side behavior relies on it. Additionally, if a service A has no associated capability but can only be called if an optional service B is called before, then service A is also considered optional. This is for instance the case for the *Cancel Work Done Progress* LSP service, which requires *Create Work Done Progress* to be called before.

We also identify services that are dependent on concrete syntax. Some of these services are tied to the concrete representation of a program through the arguments of either their request or response. More specifically, LSP and DAP were designed to work with programs relying on a textual syntax; this requirement might influence the suitability of these protocols for languages using other forms of concrete representation. Therefore, we specify for each service whether they are dependent on concrete syntax or not. A service is considered dependent on concrete syntax if it has at least one

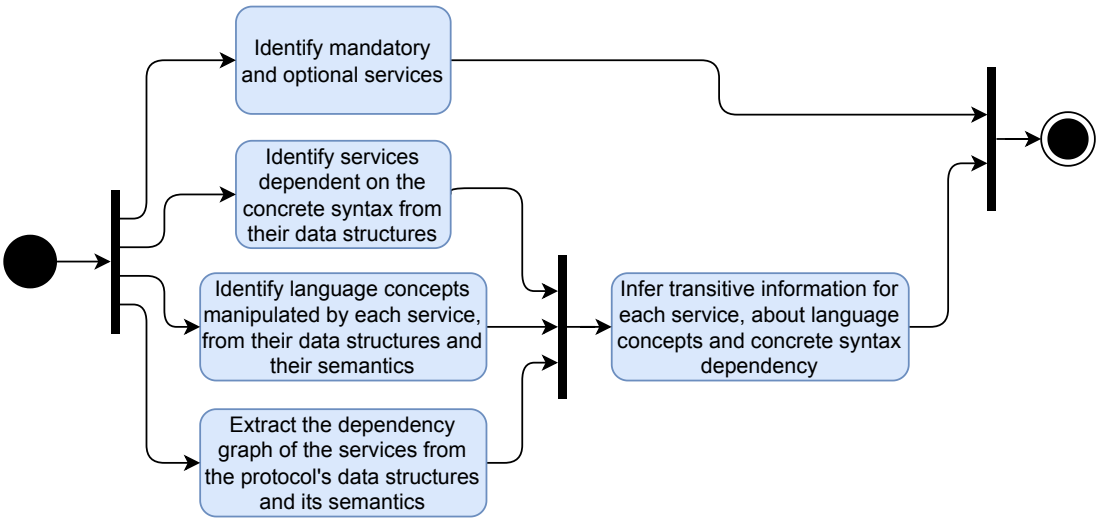

Fig. 1. Overview of the review process.

mandatory attribute related to textual syntax, such as a location expressed in terms of line and/or column, a range, or a portion of code encoded as a string. It is also considered dependent on concrete syntax if its semantics clearly mention such dependency, e.g. the text document synchronization services of LSP. A service is considered partially dependent on concrete syntax if the only arguments related to textual syntax are either optional attributes, or mandatory attributes that have a default value that will result in them being ignored. Finally, a service is considered agnostic of concrete syntax if none of its attributes are related to textual syntax.

In addition, we identify the concepts that are required from a language in order for this language to be able to implement each service. For instance, in the LSP evaluation, there is a group of services that require no specific language concept; there is another group for services that require the presence of callable elements, such as functions. A service can require a language concept for multiple reasons: first, a mandatory attribute referencing the required concept is present in the request or response of the service. As such, the *StackTrace* DAP service requires the notion of threads since this service takes a thread ID as an attribute in its arguments. Second, the required concept is mandated by the semantics of the service. For instance, the DAP service *StepInTargets* does not explicitely mention functions in its arguments. However, the only purpose of this service is to return targets that the *StepIn* service can optionally use to specify a target to step in.

Furthermore, we draw a graph of the dependencies between the services of each protocol. A service A is dependent on another service B if B must be called before A. We determine these dependencies either when they are explicitly mentioned in the semantics of the protocol, or when a service requires an argument that can only be retrieved by calling another specific service. As an example, DAP contains a number of services which require a thread ID as argument; this ID must correspond to the ID of one of the threads returned by the *Threads* service. We then consider those services to depend on the *Threads* service.

Once we have completed the last three steps, we can infer transitive information for each service based on its dependencies. The first information we infer revolves around concrete syntax. A service is considered transitively dependent on concrete syntax if itself or at least one of the services it depends on is completely dependent on concrete syntax. Otherwise, a service is considered transitively partially dependent on concrete syntax if itself or at least one of the services it depends on is partially dependent on concrete syntax (and the service is not transitively dependent on concrete syntax). The second inferred information is about required language concepts. A service transitively requires a language concept if itself or at least one of the services it depends on directly requires this concept.

Finally, when all the previous steps are achieved, we simply aggregate all the previously gathered results in a complete classification of services.

B. Language Server Protocol

This review is based on version 3.17 of LSP[3]. The specification of LSP already provides a set of categories for its different services:

- *Lifecycle Messages*: Services related to the start and shutdown of the language server by the client, as well as the declaration of capabilities.
- *Document Synchronization*: Services related to the synchronization between the state stored by the language server and the state of the text (or notebook) document.
- *Workspace Features*: Services related to workspace-wide operations, such as file creation, folder navigation, etc...
- *Window Features*: Services related to window-specific operations, such as logging or progress tracking.

[3]https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

- *Language Features*: Services related to language-specific operations.

We review LSP services from all the aforementioned categories.

1) Identified Language Concepts: In this section, we describe the language concepts identified during our review of the aforementioned LSP services.

Executability: These services are only relevant to languages with execution semantics. More specifically, the language should be able to evaluate part of a program and return a value that can then be manipulated by the UI.

Element Referencing: These services are relevant to languages that can declare elements and reference them at other places in the code through a name binding. For instance, there are a number of *Go to* operations that bring the focus of the editor to either the declaration, definition or implementation of a specific element.

Type Definition: These services only fit languages that allow custom type definition. This type definition can be as simple as declaring an alias for a base type, or propose more advanced features such as fields definition or even operations definition as in object-oriented programming.

Callable Elements: These services require the definition of elements (e.g. functions) that can be called from other places in the code.

Resource Linking: These services need the language to support links to external resources, such as a website or another file.

Import / Export: These services requires the language to support the import and export of source code from other files, either partially or entirely.

Color Referencing: These services are relevant to languages where color can be referenced. They are relevant for a UI to directly display a color next to its reference, or even present the user with a color picker.

2) Services Dependencies: Fig. 2 shows the dependency graph of the considered LSP services. We can identify multiple dependencies that follow the same patterns.

First, there exist multiple *Refresh* services; they simply trigger a new call to another target service. As such, the target service must be implemented in order for the associated *Refresh* service to have any meaningful effect.

Another group of services involves *Prepare* services; they perform some preliminary task in order for their target service to be called.

Finally, *Resolve* services compute additional information for a response obtained from a previous call to another service.

There also exist a slightly more complex dependency structure for text document synchronization, during which an order must be respected.

It is interesting to note that there exists a single explicit dependency between the two protocols. The *Inline Value* LSP services computes the value of an expression in a given context, which can then be displayed in the editor. The context that is passed to the language during this request directly references a stack frame as defined in DAP.

3) Results: Table I presents the results of our review for the *Language Features* LSP services, following the process presented at the beginning of this section. The first column lists all the evaluated services, taken from the *Language Features* group presented in the specification of the protocol. The second column highlights whether the service is provided by the language server or UI. The third and fourth columns reflect the concepts a language must manipulate, either directly or indirectly, in order to be able to implement the service specified in the first column. Transitive dependencies on such concepts are the union of direct and indirect dependencies. Following the same logic, the fifth and sixth columns provide information about the direct and transitive dependency of a service to concrete syntax. Finally, the last column determines whether a service is mandatory or not.

Table II presents the result of our review for the remaining LSP services. It follows the same structure as Table I, but contains an additional column specifying the category to which a service belongs.

C. Debug Adapter Protocol

This review is based on version 1.61 of the protocol[4].

1) Identified Language Concepts: In this section, we describe the language concepts identified during our review of DAP services. While not explicitely mentioned in the presented data, we consider that only executable languages can benefit from DAP.

Steps: These services can be implemented by languages with a notion of steps. Steps are transitions between coherent runtime states of a program. It can be argued that every executable language can define steps. However, for a language with only a single step in each program, services that require multiple steps have no interest.

Expressions: These services only work if a language supports expressions, i.e. part of a program that can be individually evaluated.

Scopes: Languages must support scoping in order to implement these services. Scopes can be defined as a portion of code in which a name binding is valid.

Threads: These services expect the language to be able to list the threads a program is currently being executed on.

Stack: To implement these services, a language must manipulate concepts related to an execution stack, such as stack traces or frames.

Variables: These services are relevant for languages allowing the use of variables. Variables are very broadly defined in DAP as a key/value pair. These pairs can be used to represent arbitrary concepts in a language, but are commonly used for properties, methods, classes, etc.

Modules: These services can be implemented by languages that can use modules, i.e. external executable files.

Disassembly: These services are relevant for languages whose source code can be translated to disassembled instructions (e.g. machine code or bytecode).

[4]https://microsoft.github.io/debug-adapter-protocol/specification

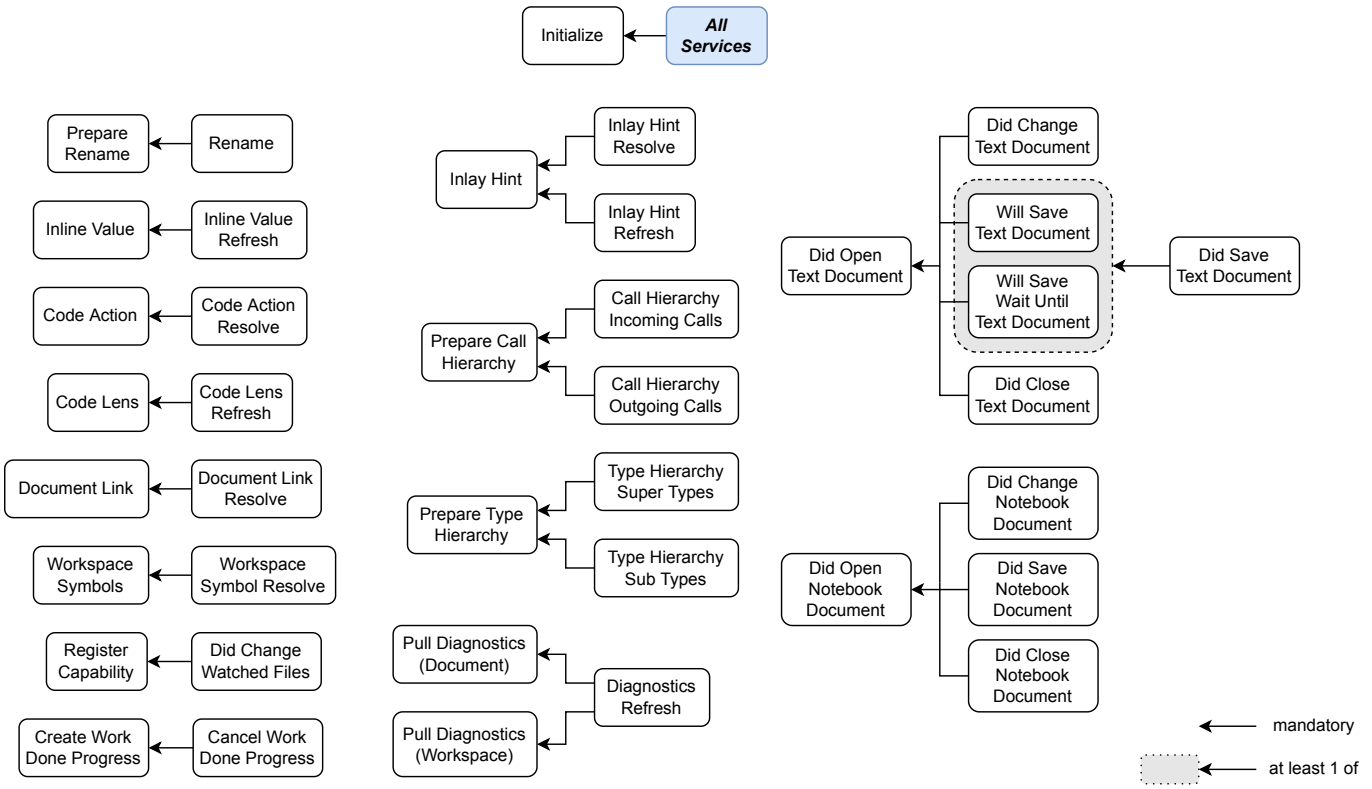

Fig. 2. Dependency graph for LSP services.

Memory Access: Only languages that can directly read bytes from and write bytes to the memory can implement these services.

Functions: These services are relevant for languages allowing the use of functions.

Exceptions: These services are relevant for languages with support for exception throwing, i.e. an event halting the normal execution of a program. It also presents a specific data structure holding more detailed information. When unhandled, such event results in the brutal stop of the running program, but most languages offer structures to gracefully process exceptions in code without interrupting the execution.

Backwards Execution: These services are meant for languages that support backwards execution. For a language to support backwards execution, each operation contained in said language must be reversible in some way, e.g. possibility to undo the operation or to roll back to the state before said operation. Supporting backwards execution may be especially difficult for languages with side effects, such as message exchanges or database modifications.

2) Services Dependencies: Fig. 3 shows the dependency graph of the DAP services. The central services that can be identified are the ones called to fill the variables view of the UI: *Threads*, *StackTrace*, *Scopes* and *Variables*. *DataBreakpointInfo* has an optional dependency to *Variables* because it relies either on **Expressions** or **Variables** language concepts. Since expressions are simply encoded as strings in this service, there is no dependency to other services if *DataBreakpointInfo* relies on **Expressions**. If it relies on **Variables** however, it must reference one via an identifier given as a response of the *Variables* service. *SetInstructionBreakpoints*[5] must be passed either a reference to a disassembled instruction or a memory reference. These references can be obtained through different services, one of which must be implemented in order to support *SetInstructionBreakpoints*. Notifications for the update or end of a progress must contain the ID of the corresponding progress, created in *ProgressStart*. Finally, a *Stopped* notification can only happen if the execution was unfolding, which can be the result of different services.

3) Results: Table III presents the results of our review of DAP, following the process presented at the beginning of this section. The first column lists all the evaluated services, which are all the requests listed in the specification. The second column highlights whether the service is provided by the debug adapter or the UI. The third and fourth columns reflect the concepts a language must manipulate, either directly or indirectly, in order to be able to implement the service specified in the first column. Transitive dependencies are the union of direct and indirect dependencies. The fifth and sixth columns provide information about the direct and transitive dependency of a service to concrete syntax. Finally, the last column determines whether a service is mandatory or not.

[5]Not to be confused with the *SetBreakpoints* service that applies breakpoints to the program's source code, whereas *SetInstructionBreakpoints* works on underlying disassembled instructions.

TABLE I
CLASSIFICATION OF *Language Features* LSP SERVICES.
●: YES, ○: NO, ∼: PARTIALLY
UI: USER INTERFACE, LS: LANGUAGE SERVER

Service	Provided By	Required Language Concepts		Concrete Syntax Dependent		Mandatory
		Direct	*Indirect*	*Direct*	*Transitive*	
Hover	LS	∅	∅	●	●	○
Completion Proposals	LS	∅	∅	●	●	○
Completion Item Resolve	LS	∅	∅	●	●	○
Pull Diagnostics (Document)	LS	∅	∅	●	●	○
Pull Diagnostics (Workspace)	LS	∅	∅	●	●	○
Folding Range	LS	∅	∅	●	●	○
Selection Range	LS	∅	∅	●	●	○
Formatting	LS	∅	∅	●	●	○
On type Formatting	LS	∅	∅	●	●	○
Range Formatting	LS	∅	∅	●	●	○
Linked Editing Range	LS	∅	∅	●	●	○
Inlay Hint	LS	∅	∅	●	●	○
Inlay Hint Resolve	LS	∅	∅	●	●	○
Code Lens	LS	∅	∅	●	●	○
Document Symbols	LS	∅	∅	●	●	○
Semantic Tokens (Full)	LS	∅	∅	●	●	○
Semantic Tokens (Full Delta)	LS	∅	∅	●	●	○
Semantic Tokens (Range)	LS	∅	∅	●	●	○
Code Action	LS	∅	∅	●	●	○
Code Action Resolve	LS	∅	∅	∼	●	○
Publish Diagnostics	UI	∅	∅	●	●	○
Diagnostics Refresh	UI	∅	∅	○	●	○
Inlay Hint Refresh	UI	∅	∅	○	●	○
Code Lens Refresh	UI	∅	∅	○	●	○
Semantic Tokens Refresh	UI	∅	∅	○	●	○
Inline Value	LS	Executability	∅	●	●	○
Inline Value Refresh	UI	Executability	∅	●	●	○
Go to Declaration	LS	Element Referencing	∅	●	●	○
Go to Definition	LS	Element Referencing	∅	●	●	○
Go to Implementation	LS	Element Referencing	∅	●	●	○
Find References	LS	Element Referencing	∅	●	●	○
Document Highlight	LS	Element Referencing	∅	●	●	○
Rename	LS	Element Referencing	∅	●	●	○
Prepare Rename	LS	Element Referencing	∅	●	●	○
Prepare Type Hierarchy	LS	Type Definition	∅	●	●	○
Type Hierarchy Super Types	LS	Type Definition	∅	●	●	○
Type Hierarchy Sub Types	LS	Type Definition	∅	●	●	○
Prepare Call Hierarchy	LS	Callable Elements	∅	●	●	○
Call Hierarchy Incoming Calls	LS	Callable Elements	∅	●	●	○
Call Hierarchy Outgoing Calls	LS	Callable Elements	∅	●	●	○
Signature Help	LS	Callable Elements	∅	●	●	○
Document Link	LS	Resource Linking	∅	●	●	○
Document Link Resolve	LS	Resource Linking	∅	●	●	○
Moniker	LS	Import / Export	∅	●	●	○
Document Color	LS	Color Referencing	∅	●	●	○
Color Presentation	LS	Color Referencing	∅	●	●	○
Go to Type Definition	LS	Element Referencing ∧ Type Definition	∅	●	●	○

IV. DISCUSSION

In this section, we further discuss the implications of the results highlighted in Section III about the suitability of both LSP and DAP for DSL tooling. Tables IV and V respectively present an aggregation of numerical results for our review of LSP and DAP.

We identify two reasons that may hinder the implementation of certain services for some DSLs: required language concepts and concrete syntax dependency. The first aspect refers to the fact that different services require the language to manipulate specific concepts, which may not all be supported by every DSL. Regarding LSP, 79% the services require no distinctive concepts. However, this concerns only about half of the services related to *Language Features*. For DAP, this part drops to 38%. In addition, if mandatory services of a protocol rely on specific concepts, then any DSL needs to include those concepts to be able to use the protocol. For LSP, the identified mandatory services do not require any distinctive concept. DAP however defines numerous mandatory services, together relying on the concepts of **Steps**, **Threads**, **Stack**, **Scopes**, **Variables**, **Expressions** and **Functions**.

The second aspect concerns the dependency of some ser-

TABLE II
CLASSIFICATION OF OTHER LSP SERVICES.
●: YES, ○: NO, ∼: PARTIALLY
UI: USER INTERFACE, LS: LANGUAGE SERVER

Category	Service	Provided By	Required Language Concepts		Concrete Syntax Dependent		Mandatory
			Direct	*Indirect*	*Direct*	*Transitive*	
Lifecycle Messages	Initialize	LS	∅	∅	●	●	●
	Initialized	LS	∅	∅	○	●	●
	Set Trace	LS	∅	∅	○	●	●
	Shutdown	LS	∅	∅	○	●	●
	Exit	LS	∅	∅	○	●	●
	Log Trace	UI	∅	∅	○	●	●
	Register Capability	UI	∅	∅	○	●	○
	Unregister Capability	UI	∅	∅	○	●	○
Document Synchronization	Did Open Text Document	LS	∅	∅	●	●	○
	Did Change Text Document	LS	∅	∅	●	●	○
	Will Save Text Document	LS	∅	∅	●	●	○
	Will Save Wait Until Text Document	LS	∅	∅	●	●	○
	Did Save Text Document	LS	∅	∅	●	●	○
	Did Close Text Document	LS	∅	∅	●	●	○
	Did Open Notebook Document	LS	∅	∅	●	●	○
	Did Change Notebook Document	LS	∅	∅	●	●	○
	Did Save Notebook Document	LS	∅	∅	●	●	○
	Did Close Notebook Document	LS	∅	∅	●	●	○
Workspace Features	Did Change Configuration	LS	∅	∅	○	●	●
	Workspace Symbols	LS	∅	∅	○	●	○
	Workspace Symbol Resolve	LS	∅	∅	○	●	○
	Did Change Workspace Folders	LS	∅	∅	○	●	○
	Will Create Files	LS	∅	∅	○	●	○
	Did Create Files	LS	∅	∅	○	●	○
	Will Rename Files	LS	∅	∅	○	●	○
	Did Rename Files	LS	∅	∅	○	●	○
	Will Delete Files	LS	∅	∅	○	●	○
	Did Delete Files	LS	∅	∅	○	●	○
	Did Change Watched Files	LS	∅	∅	○	●	○
	Execute Command	LS	∅	∅	○	●	○
	Apply Edit	UI	∅	∅	∼	●	○
	Configuration	UI	∅	∅	○	●	○
	Workspace Folders	UI	∅	∅	○	●	○
Window Features	Cancel Work Done Progress	LS	∅	∅	○	●	○
	Log Message	UI	∅	∅	○	●	●
	Show Message (Notification)	UI	∅	∅	○	●	●
	Telemetry	UI	∅	∅	○	●	●
	Show Message (Request)	UI	∅	∅	○	●	○
	Show Document	UI	∅	∅	○	●	○
	Create Work Done Progress	UI	∅	∅	○	●	○

vices on the concrete textual syntax. While this type of concrete representation is popular, there exist other kinds of syntaxes that DSLs may rely on, such as graphical or projectional. All LSP services are indirectly dependent on concrete syntax since they all rely on the *Initialize* service, which is itself dependent on the concrete syntax (notably to set the encoding of documents). This makes sense, as the protocol was explicitly developed for textual editing, but one can question whether some commonalities can be found with other editing protocols such as GLSP [6]. An increase in the number of editing protocols means that it becomes more costly for language engineers to interface with all of them. For a language with both a textual and graphical syntax, implementing both the LSP and GLSP interface may induce some redundancy. It may be interesting to explore whether an updated version of LSP could factorize services to support editing for different kinds of concrete syntaxes. DAP only has 25% of its services directly relying (completely or partially) on concrete syntax. However, since all DAP services depend on the *Initialize* service, which is itself dependent on the concrete syntax, then all services are indirectly dependent on concrete syntax. This mandatory dependency to concrete syntax through the *Initialize* service can be mitigated by simply ignoring the proper attributes of this request on the language side (that is, the attributes describing the UI's offset for lines and columns), but this requires to stray from the original semantics of the protocol, a strategy discussed later on.

If for one of these reasons a DSL is not able to provide a service, this will have a different impact depending on the way this DSL implements the protocols. If the implementation strictly conforms to the specification, then not implementing a service will not have any negative effect as long as it is an optional service. Obviously, if a DSL decides to provide a service that depends on other services, then those additional

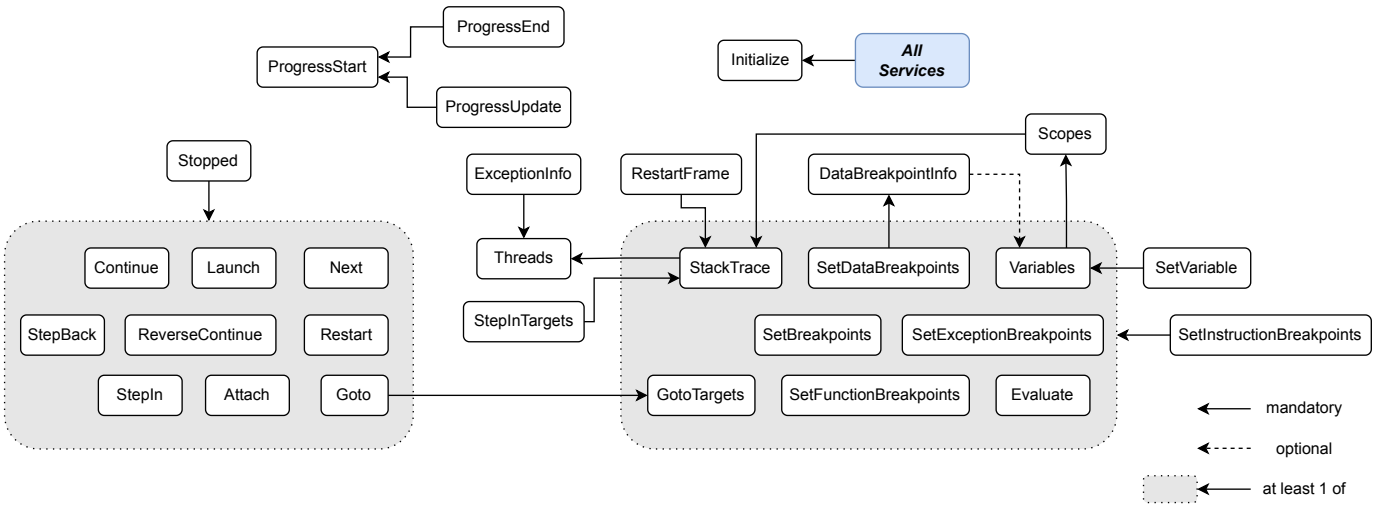

Fig. 3. Dependency graph for DAP services.

services must also be implemented. If a mandatory service is not provided, then unpredictable effects may occur within the UI. On the other hand, if the DSL implements LSP or DAP in a looser way, it might be able to provide services which may seem impossible to implement when conforming strictly to the specification. One strategy is to use mock values to fake support for some language concepts. For instance, support for threads in DAP may be mocked by pretending to have a single thread returned by the *Threads* service, and then referenced in other services requiring threads. Still, this approach may produce unpredictable results within the UI, as it treats the returned values according to the specification. For instance, for a language that does not manipulate the notion of thread, providing a single mock thread in order to fill the variables view will still result in this thread being displayed as a ”real” thread, which may cause some confusion for the end-user. Another strategy is to provide a mapping between concepts present in the language to concepts manipulated by the protocol. For example, a DSL can use the *Variables* DAP service to communicate values other than variables as usually understood in GPLs, such as the states of a state machine as implemented in [8]. However, these values may still be manipulated as such by the UI and still be referenced as variables.

It is interesting to note that through the *Code Action* service, LSP provides a certain level of flexibility that allows language engineers to define domain-specific editing operations. For instance, the ”Remove State” operation presented in Section II could be implemented using this mechanism. Still, not all operations can be added this way: as an example, complex workflows that do not consist of a single request to the language server are not supported by this approach, as it would require to change the implementation of LSP in UIs in order to properly handle the additional requests and / or computation. In contrast, DAP does not provide any mechanism to easily configure new domain-specific debugging operations. For instance, the PetitParser DSL mentioned in Section II cannot rely on DAP to provide debugging operations related to parsing rules, production or input matching.

We can also point out a strong imbalance between LSP and DAP in terms of service coupling, as clearly shown by comparing Fig. 2 and 3. The evaluated LSP services have a maximum of two degree of dependency, i.e. they transitively depend on a maximum of two services. In the other hand, DAP presents an overall more complex structure, containing services with higher degrees of dependency or having optional dependencies. Because of this, mocking or mapping DSL concepts for this protocol can have complex repercussions on other services.

V. Related Work

Barros et al. [9] mined through existing implementations of LSP by languages and extracted common practices. The authors do broadly classify the languages they work with—using terms such as ”imperative”, ”declarative” or ”functional”—and explore the correlation between these paradigms and implemented services. However, they do not provide an in-depth analysis of the underlying fine-grained concepts present in each language. In the same way, the authors also highlight services that are often implemented together, but do not rely on the specification of the protocol.

Bünder et al. [10] review the integration of LSP for XText projects by performing a case study and conduct a SWOT analysis for the XText LSP integration, supported by the mining of multiple existing projects. This work does not provide any classification of the examinated DSLs, and offers no discussion about the LSP protocol itself.

Jeanjean et al. [11] propose a vision where the features of IDEs are dynamically configured by the languages they are currently working with. LSP and DAP enable some level of configurability, especially through *capabilities*. However, as discussed earlier, their configurability is limited and would

TABLE III
CLASSIFICATION OF DAP SERVICES.
●: YES, ○: NO, ∼: PARTIALLY
UI: USER INTERFACE, DA: DEBUG ADAPTER

Service	Provided By	Required Language Concepts		Concrete Syntax Dependent		Mandatory
		Direct	*Indirect*	*Direct*	*Transitive*	
Initialize	DA	∅	∅	●	●	●
Source	DA	∅	∅	●	●	●
Launch	DA	∅	∅	○	●	●
Attach	DA	∅	∅	○	●	●
Disconnect	DA	∅	∅	○	●	●
Cancel	DA	∅	∅	○	●	●
Completions	DA	∅	∅	●	●	○
ConfigurationDone	DA	∅	∅	○	●	○
Restart	DA	∅	∅	○	●	○
Terminate	DA	∅	∅	○	●	○
LoadedSources	DA	∅	∅	○	●	○
Output	UI	∅	∅	●	●	●
Process	UI	∅	∅	○	●	●
Initialized	UI	∅	∅	○	●	●
Exited	UI	∅	∅	○	●	●
Terminated	UI	∅	∅	○	●	●
LoadedSource	UI	∅	∅	○	●	●
Capabilities	UI	∅	∅	○	●	●
ProgressStart	UI	∅	∅	○	●	○
ProgressUpdate	UI	∅	∅	○	●	○
ProgressEnd	UI	∅	∅	○	●	○
RunInTerminal	UI	∅	∅	○	●	○
StartDebugging	UI	∅	∅	○	●	○
SetBreakpoints	DA	Steps	∅	●	●	●
BreakpointLocations	DA	Steps	∅	●	●	○
GotoTargets	DA	Steps	∅	●	●	○
Breakpoint	UI	Steps	∅	○	●	●
Threads	DA	Threads	∅	○	●	●
TerminateThreads	DA	Threads	∅	○	●	○
Stopped	UI	Threads	∅	○	●	●
Continued	UI	Threads	∅	○	●	●
Thread	UI	Threads	∅	○	●	●
Invalidated	UI	Threads	∅	○	●	○
RestartFrame	DA	Stack	Threads	○	●	○
Variables	DA	Variables	Scopes ∧ Stack ∧ Threads	○	●	●
SetVariable	DA	Variables	Scopes ∧ Stack ∧ Threads	○	●	○
Modules	DA	Modules	∅	○	●	○
Module	UI	Modules	∅	○	●	●
Disassemble	DA	Disassembly	∅	∼	●	○
ReadMemory	DA	Memory Access	∅	○	●	○
WriteMemory	DA	Memory Access	∅	○	●	○
Memory	UI	Memory Access	∅	○	●	○
Evaluate	DA	Expressions ∧ Scopes	∅	○	●	●
SetExpression	DA	Expressions ∧ Scopes	∅	○	●	○
ExceptionInfo	DA	Exceptions ∧ Threads	∅	○	●	○
StackTrace	DA	Stack ∧ Threads	∅	∼	●	●
Scopes	DA	Stack ∧ Scopes	Threads	∼	●	●
Pause	DA	Steps ∧ Threads	∅	○	●	●
Next	DA	Steps ∧ Threads	∅	○	●	●
Continue	DA	Steps ∧ Threads	∅	○	●	●
Goto	DA	Steps ∧ Threads	∅	○	●	○
SetInstructionBreakpoints	DA	Steps ∧ Disassembly	∅	∼	●	○
SetFunctionBreakpoints	DA	Steps ∧ Functions	∅	∼	●	○
SetExceptionBreakpoints	DA	Steps ∧ Exceptions	∅	∼	●	○
SetDataBreakpoints	DA	Steps ∧ (Expressions ∨ Variables)	∅ ∨ (Threads ∧ Stack ∧ Scopes)	∼	●	○
DataBreakpointInfo	DA	Steps ∧ (Expressions ∨ Variables)	∅ ∨ (Threads ∧ Stack ∧ Scopes)	○	●	○
StepBack	DA	Steps ∧ Threads ∧ Backwards Execution	∅	○	●	○
ReverseContinue	DA	Steps ∧ Threads ∧ Backwards Execution	∅	○	●	○
StepIn	DA	Steps ∧ Threads ∧ Functions	∅ ∨ Stack	○	●	●
StepOut	DA	Steps ∧ Threads ∧ Functions	∅	○	●	●
StepInTargets	DA	Steps ∧ Stack ∧ Functions	Threads	∼	●	○

TABLE IV
NUMERICAL RESULTS AGGREGATION FOR THE LSP EVALUATION

Metric		Number of affected services
Evaluated Services		87
Mandatory Services	*Mandatory*	10 (11%)
	Optional	77 (89%)
Concrete Syntax Dependent Services (Transitive)	*Yes*	87 (100%)
	Partially	0 (0%)
	No	0 (0%)
Required Language Concepts (Transitive)	$\emptyset$	69 (79%)
	Element Referencing	8 (9%)
	Type Definition	4 (5%)
	Callable Elements	4 (5%)
	Executable	2 (2%)
	Resource Linking	2 (2%)
	Color Referencing	2 (2%)
	Import / Export	1 (1%)

TABLE V
NUMERICAL RESULTS AGGREGATION FOR THE DAP EVALUATION

Metric		Number of affected services
Evaluated Services		61
Mandatory Services	*Mandatory*	29 (48%)
	Optional	32 (52%)
Concrete Syntax Dependent Services (Transitive)	*Yes*	61 (100%)
	Partially	0 (0%)
	No	0 (0%)
Required Language Concepts (Transitive)	$\emptyset$	23 (38%)
	Threads	21-23 (34-38%)
	Steps	18 (30%)
	Stack	6-9 (10-15%)
	Scopes	5-7 (8-11%)
	Functions	4 (7%)
	Memory Access	3 (5%)
	Expressions	2-4 (3-7%)
	Variables	2-4 (3-7%)
	Disassembly	2 (3%)
	Exceptions	2 (3%)
	Backwards Execution	2 (3%)
	Modules	2 (3%)

benefit in this regard from the approach proposed in the aforementioned paper.

Multiple papers also present the work of interfacing given languages with LSP or DAP. Bour et al. [12] report on their experience of implementing LSP support for OCaml, a functional programming language. Sander [13] describe the implementation of LSP services for the Nickel configuration language. Our work is based on a careful inspection of the specification of both LSP and DAP, and does not rely on existing implementations by DSLs.

Finally, some papers propose an extension to these protocols to support new features related to a given domain. Karmios et al. [14] extend DAP to support operations related to symbolic execution, and apply this new protocol to integrate a debugger within the Gillian platform. Ernst et al. [15] specify new semantics for a subset of DAP to enable support for deductive verification, and apply this specification to implement a debugger for SecC, a program verifier for C. Rask et al. [16] provide an extension to LSP in order to support additional operations for specification languages, such as code generation or theorem proving. In this paper, we describe the purpose, properties and limitations of existing LSP and DAP services, but do not propose any extension nor modification to take advantage of other language concepts.

VI. CONCLUSION

In this work, we proposed a qualitative review of both LSP and DAP, based on an analysis of their respective specification. This review allowed us to provide an overview of the functioning of these protocols, as well as a more fine-grained analysis at the service level. We identified some key aspects that limit the ability of DSLs to interface with LSP and DAP. Most notably, a number of services rely on specific language concepts that may not be part of every DSL. On the other side, additional concepts manipulated by DSLs cannot be taken advantage of through these protocols to provide related domain-specific operations.

Multiple research perspectives are highlighted by this work:

- Is it possible to decouple concrete syntax from the rest of the protocol in LSP and DAP? This would make it easier to integrate new paradigms, such as graphical or projectional syntaxes, and would limit the appearance of new, concurrent protocols like GLSP[6].
- An empirical study about the current practices to implement LSP and DAP for DSLs could be performed on existing tools. This study could highlight common strategies to deal with the bias present in these protocols. This differs from [9], where the authors look at which LSP services are implemented by different languages, but do not discuss the changes to the original LSP semantics that were made in order to implement those services.
- Finally, it would be interesting to reflect on how to use those protocols to provide domain-specific operations for a broader panel of languages. This work was already started in [8], which proposes a new architecture for creating configurable interactive debuggers for DSLs, applied to the support of domain-specific breakpoints. This work can be extended to include other domain-specific debugging operations—such as domain-specific steps or configuration of the variables view—and to support other execution paradigms like non-determinism and parallelism.

REFERENCES

[1] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org, 2013.

[2] Microsoft, "Official page for Language Server Protocol," https://microsoft.github.io/language-server-protocol/, 2023.

[3] ——, "Official page for Debug Adapter Protocol," https://microsoft.github.io/debug-adapter-protocol/, 2023.

[4] A. Zeller, "How debuggers work," in *The Debugging Book*. CISPA Helmholtz Center for Information Security, 2023, retrieved 2023-01-06 17:58:51+01:00. [Online]. Available: https://www.debuggingbook.org/html/Debugger.html

[6] https://www.eclipse.org/glsp/

[5] A. Chiş, T. Gîrba, and O. Nierstrasz, "The moldable debugger: A framework for developing domain-specific debuggers," in *International Conference on Software Language Engineering*. Springer, 2014, pp. 102–121.

[6] Eclipse Fundation, "GLSP," https://www.eclipse.org/glsp/.

[7] Build Server Protocol, "Build Server Protocol," https://build-server-protocol.github.io/.

[8] J. Enet, E. Bousse, M. Tisi, and G. Sunyé, "Protocol-Based Interactive Debugging for Domain-Specific Languages." *The Journal of Object Technology*, vol. 22, no. 2, p. 2:1, 2023.

[9] D. Barros, S. Peldszus, W. K. G. Assunção, and T. Berger, "Editing Support for Software Languages: Implementation Practices in Language Server Protocols," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, 2022, pp. 232–243.

[10] H. Bünder and H. Kuchen, "Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol," in *Model-Driven Engineering and Software Development*, ser. Communications in Computer and Information Science, S. Hammoudi, L. F. Pires, and B. Selić, Eds. Cham: Springer International Publishing, 2020, pp. 225–245.

[11] P. Jeanjean, B. Combemale, and O. Barais, "IDE as Code: Reifying Language Protocols as First-Class Citizens," in *ISEC 2021: 14th Innovations in Software Engineering Conference, Bhubaneswar, Odisha, India, February 25-27, 2021*, D. P. Mohapatra, S. Mishra, T. Clark, A. Dubey, R. Sharma, and L. Kumar, Eds. ACM, 2021, pp. 23:1–23:5.

[12] F. Bour, T. Refis, and G. Scherer, "Merlin: A language server for OCaml (experience report)," *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, pp. 1–15, 2018.

[13] Y. Sander, "Design and Implementation of the Language Server Protocol for the Nickel Language," 2022.

[14] N. Karmios, S.-É. Ayoun, and P. Gardner, "Symbolic Debugging with Gillian," in *Proceedings of the 1st ACM International Workshop on Future Debugging Techniques*, 2023, pp. 1–2.

[15] G. Ernst, J. Blau, and T. Murray, "Deductive Verification via the Debug Adapter Protocol," *arXiv preprint arXiv:2108.02968*, 2021.

[16] J. K. Rask, F. P. Madsen, N. Battle, H. D. Macedo, and P. G. Larsen, "The specification language server protocol: A proposal for standardised LSP extensions," *arXiv preprint arXiv:2108.02961*, 2021.