Cross-IDE remote debugging of model management programs through the Debug Adapter Protocol

Antonio García-Domínguez
a.garcia-dominguez@york.ac.uk
University of York
York, United Kingdom

Dimitris Kolovos
dimitris.kolovos@york.ac.uk
University of York
York, United Kingdom

Abstract

Eclipse Epsilon is an open-source family of model management languages and tools, which has seen significant use in industry and academia. Epsilon programs have been used in a variety of scenarios, from being simply run in the Eclipse IDE, to being embedded in Eclipse plugins, Java programs, web services, Ant workflows, and Gradle build scripts. When one of these embedded Epsilon programs showed unexpected behaviour, debugging it required running it from the Eclipse IDE: reproducing the behaviour was complicated if it also required recreating a complex environment. Likewise, users asked for supporting debugging from other IDEs beside Eclipse, as its market share has dropped in the last years. In this demo, we will show a new feature in Epsilon 2.6 which allows for remote debugging of Epsilon programs in a broader range of scenarios, using the Microsoft Debug Adapter Protocol. We will also demonstrate how this remote debugging support can be reused from other IDEs (specifically, Microsoft Visual Studio Code), with minimal effort compared to re-implementing a dedicated debugger.

CCS Concepts

• **Software and its engineering → Integrated and visual development environments**; *Domain specific languages.*

Keywords

Remote debugging, Debug Adapter Protocol, Eclipse, Visual Studio Code, Epsilon, model management languages

ACM Reference Format:
Antonio García-Domínguez and Dimitris Kolovos. 2024. Cross-IDE remote debugging of model management programs through the Debug Adapter Protocol. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3652620.3687783

1 Introduction

The Eclipse Epsilon family of model management languages [12] has seen significant use across industry[1] (having been embedded inside commercial solutions such as Rolls-Royce's CaMCoA Studio [2]) and education[2]. As its userbase has broadened, demand has grown for debugging Epsilon programs running in a variety of environments and from different integrated development environments (IDEs).

Epsilon programs are often executed outside of an IDE: they can be run from Ant and Gradle workflows, and from programs written in JVM-compatible languages (as a form of scripting for model management). These embedded Epsilon programs may need debugging as well. Up to the current stable version of Epsilon (2.5), debugging support was limited to programs running from Eclipse launch configurations: this meant that users of embedded Epsilon programs had to find a way to reproduce their setup from Eclipse, which was non-trivial in scenarios with significant customisations to the environment.

Additionally, as core developers of Eclipse Epsilon, we have seen increased demand for improved support in other IDEs besides Eclipse. Recent Stack Overflow developer surveys [19] have shown that Eclipse's market share has been steadily declining: while 15.87% of the 82,277 responses used Eclipse in 2021, this dropped to 12.57% out of 71,010 responses in 2022 and then 9.9% out of 86,544 responses in 2023. A community member developed a Visual Studio Code extension [11] (the most popular IDE in the 2023 Stack Overflow survey, selected by 73.71% of the responses) to provide syntax highlighting for Epsilon programs, with an embedded language server for syntactic validation. While it was possible to reuse the existing Gradle and Maven support in VS Code to run Epsilon programs through Epsilon's Ant tasks, the debugging facilities present in Eclipse were not available.

We present a new feature that will be incorporated in the upcoming 2.6 release of Eclipse Epsilon: support for remote debugging of Epsilon programs through the Microsoft Debug Adapter Protocol (DAP) [16]. This new feature addresses the two problems above: it allows for debugging Epsilon programs running outside of an IDE, and reduces the effort involved in supporting debugging from other IDEs besides Eclipse. Specifically, the demonstration will show how remote debugging can be performed from the Eclipse and VS Code IDEs, obtaining consistent user experiences thanks to the use of the same debugging codebase.

The rest of the paper is structured as follows: Section 2 presents background knowledge, in the form of an overview of DAP and its currently available implementations. Section 3 discusses the tool to be demonstrated: the new DAP-compliant debug adapter in Epsilon. Section 4 presents related work around remote debugging, with a specific focus on its support across model management languages.

[1] https://eclipse.dev/epsilon/users/

[2] https://eclipse.dev/epsilon/users/education/

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3687783

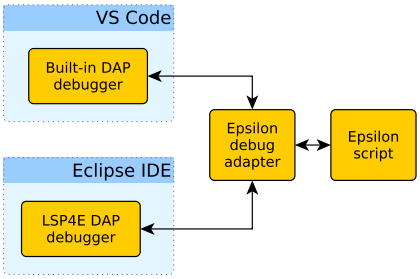

Figure 1: Communication between DAP client, DAP server, and Epsilon program

Finally, Section 5 offers some general conclusions and an overview of our plans for further development.

2 Background

This section provides a general overview of the Debug Adapter Protocol, and discusses the types of DAP-related software to consider.

2.1 The Microsoft Debug Adapter Protocol

Implementing debugging support for a language involves two aspects: i) writing the additional code needed to control the execution flow of its programs (e.g. setting breakpoints and stopping/continuing execution) and inspecting its state (e.g. listing the running threads, their stack traces, and the available variables in each stack frame), and ii) creating the appropriate tooling so the user can access these facilities. While i) ideally only needs to be developed once for the language, the traditional approach for ii) was to write new debugging extensions from scratch for every IDE, requiring significant duplication of effort.

To avoid this, Microsoft released the Debug Adapter Protocol (DAP) specification [16], which describes how a development tool (e.g. an IDE) exchanges messages back and forth with a tool-agnostic *debug adapter* that translates the specifics of the language's debugging APIs to a generic set of JSON-based messages. The goal of DAP is to enable reusing the same generic DAP-compliant debugging UI for every language that has a DAP debug adapter. This is similar in spirit to how the Microsoft Language Server Protocol (LSP) [17] aims to decouple the IDE from the details of editing programs in a specific language (e.g. syntax checking, code completion). Figure 1 shows how the DAP approach translates to Epsilon: the DAP-based debuggers in Eclipse and VS Code talk to the Epsilon debug adapter, which talks to the running Epsilon program.

DAP only requires maintaining two streams of bytes: one from the DAP client (the IDE) to the DAP server (the debug adapter), and another in reverse. These streams could come from the two sides of a TCP connection (typically used when attaching to a running program), or from the standard I/O of a process (commonly used when launching a program for debugging). DAP supports request-response communication from the client to the server (e.g. "set these breakpoints"), as well as the sending of events from the server (e.g. "the program has stopped at a breakpoint").

2.2 DAP implementations

As mentioned above, DAP essentially requires that the development tool has a compliant debugging UI, and that the language has a debug adapter. It only specifies the JSON messages and does not require using a specific set of client or server libraries: either side can be implemented in the most appropriate technical stack. This has allowed a broad variety of implementations to appear, some of which are listed in the official DAP homepage [15]. On the client side, these include desktop-based IDEs such as Eclipse (via the LSP4E [8] project) and Visual Studio Code, web-based IDEs such as Theia, and editors like Neovim. On the server side, there are debug adapters for most popular languages (e.g. JavaScript, Python, Java, C#, or C++).

In order to simplify the work involved in supporting the DAP protocol, a number of *Software Development Kits* (SDKs) have been developed by the community. Microsoft's implementations can be considered to be the reference (e.g. their TypeScript-based library for implementing and testing debug adapters). The Eclipse LSP4J [9] open-source project provides an SDK for writing debug adapters in Java, which is ideal for Epsilon as it is written in Java.

3 Tool: the Epsilon debug adapter

This section discusses the overall design of the Epsilon debug adapter, and then illustrates various usage scenarios for it. The examples are adapted from those in our sample project on GitHub[3].

3.1 Overall design

As mentioned above, the new debug adapter in Epsilon 2.6 is written on top of LSP4J, which takes care of the low-level details of message (de)serialisation, connection management, and message correlation. This allowed us to focus on the core task: to implement remote debugging for the Epsilon languages[4]. This is a total of 12 different languages, each with their own specifics for debugging: thankfully, they are all either based on a common language (the Epsilon Object Language, or EOL), or are pre-processed into that common language (templates written in the Eclipse Generation Language are transformed on-the-fly to EOL). The debug adapter mostly relies on the commonly available facilities, and delegates to the languages themselves for language-specific details (e.g. to verify whether a breakpoint location is valid or not).

Figure 2 shows a UML class diagram with a high-level view of the key components in this design. The *Program* running the Epsilon program is responsible for creating and configuring the appropriate *IEolModule* implementation (e.g. *EolModule* for EOL or *EtlModule* for ETL): since the Epsilon APIs allow for the integration of models from arbitrary technologies using its Epsilon Model Connectivty APIs [7], this has been kept away from the responsibilities of the *EpsilonDebugAdapter*. The debug adapter is therefore limited to attaching to a pre-configured Epsilon program.

Having created and set up the *IEolModule*, the *Program* then wraps the module into an instance of *EpsilonDebugServer*, which abstracts away the details of setting up the *EpsilonDebugAdapter*

[3] https://github.com/eclipse/epsilon/tree/main/examples/org.eclipse.epsilon.examples.eol.dap/epsilon

[4] Except for the Epsilon Wizard Language, which is only meaningful within a graphical modelling tool.

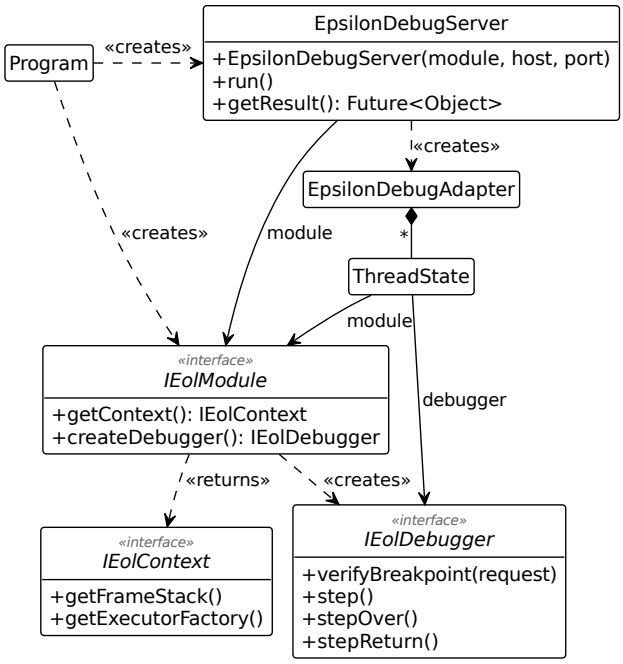

Figure 2: Simplified UML class diagram of debug adapter

and exposing it via a TCP server. The *EpsilonDebugAdapter* can track not only the initial module, but also other modules that may be launched from this initial module (e.g. the EGX template orchestration language may launch EGL templates). As shown in the figure, *EpsilonDebugAdapter* relies on generic interfaces that are implemented by all Epsilon languages, allowing it to create the appropriate *IEolDebugger* implementation and delegate to it for language-specific breakpoint verification and execution control, and to access the *IEolContext* that exposes the current frame stack and the execution control facilities of the language.

The debug adapter makes use of the Epsilon Model Connectivity layer as well. When stopped at a breakpoint, the debug adapter will delegate to the reflective capabilities of the EMC driver to inspect the properties of values that correspond to model elements. This allows it to show model element properties in the same way that they would be accessible from Epsilon programs.

3.2 Debugging Epsilon programs from Java

Having explained the high-level design, we can show the various ways in which the debug adapter can be used. If the Epsilon program is being executed from a piece of Java code, it can be changed to use remote debugging by following the approach in Listing 1. The *debug* Boolean flag is just for illustration: the exact way to choose between regular execution and remote debugging is up to the developer. When the debug server is run, it will wait for an *attach* request from a debugger before starting the Epsilon program. After the program finishes running and the server shuts down, it will get its result or throw the exception that crashed the program, as usual. It is worth noting that the *EpsilonDebugServer* class includes APIs to allow developers to map arbitrary module URIs to filesystem paths, in order to debug Epsilon programs that are loaded from

Listing 1: Running an Epsilon program in remote debugging, compared to normal execution

```
// ... module creation and setup code ...
Object result;
if (debug) {
  var server = new EpsilonDebugServer(module, port);
  server.run();
  result = server.getResult().get();
} else {
  result = module.execute();
}
// ... module disposal code ...
```

Listing 2: *launch.json* configuration for remote debugging via VS Code on port 4040

```
{
    "type": "epsilon",
    "request": "attach",
    "name": "Debug program",
    "port": 4040
}
```

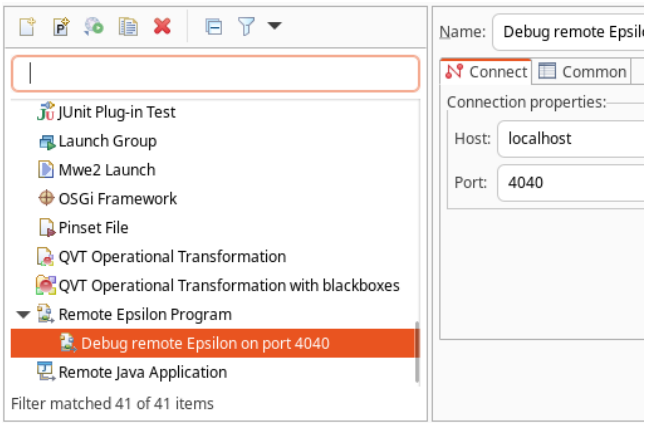

Figure 3: Configuration for remote debugging an Epsilon program on port 4040

other locations besides the local filesystem (e.g. packaged in a JAR file).

Once the debug server is waiting for connections, the developer can employ any DAP-compliant client (e.g. VS Code and its DAP-based debugger, or the Eclipse IDE after installing the LSP4E debugger). In the case of VS Code, the developer would install the community extension for Epsilon [11] and use the launch configuration in Listing 2. For Eclipse, the developer would use the new "Remote Epsilon Program" launch configuration type in Figure 3, which is based on the generic LSP4E launch configuration for debugging, with some Epsilon-specific customisations such as hyperlinks in the console for exception stack traces to the relevant code locations.

Listing 3: Debugging EOL from Gradle

```
task runEOL {
  dependsOn tasks.setupEpsilonTasks
  doLast {
    ant.'epsilon.eol'(src: 'program.eol', debug: true,
        debugPort: 4040)
  }
}
```

3.3 Debugging Epsilon programs from Ant tasks

Epsilon provides Ant tasks for workflow automation, and there was already functionality for debugging these programs so long as they were run from the same JVM as the Eclipse IDE. The new remote debugging capabilities remove this restriction, allowing for debugging Ant workflows running in a different process. A user only needs to set the *debug* option to *true* in their task:

```
<epsilon.eol ... debug="true" debugPort="4040"/>
```

Internally, the Epsilon Ant tasks rely on a *Host* interface that abstracts away the differences between running inside and outside the Eclipse IDE. The *DefaultHost* implementation is used when running outside Eclipse, and it will use the same approach as in Listing 1, where the port is specified by the *debugPort* task option: the user will need to separately launch a DAP debugger that connects to the same port. When running inside the Eclipse JVM, the *EclipseHost* implementation will be used, which will programmatically create and launch a "Remote Epsilon Program" launch configuration similar to that in Figure 3: in this case, *debugPort* can be omitted to allow the server to choose any available TCP port from an ephemeral range, which will be used in the launch configuration.

3.4 Debugging Epsilon programs from Gradle in VS Code

Given that Gradle can reuse Ant tasks and that VS Code has strong support for Gradle via an extension, the approach in Section 3.3 can be adapted to debug Epsilon programs running via Gradle builds[5]. Using Gradle over Ant has a number of advantages, namely in its built-in dependency management, and the greater expressiveness of Groovy/Kotlin compared to XML.

Starting from a Gradle buildfile such as the one in Epsilon's documentation[6], the Gradle task to debug an EOL program would look as in Listing 3. The EOL task is part of a Gradle *doLast* block, to ensure that it is only run when explicitly invoked, and not in the Gradle *configure* phase used by IDEs to list the available tasks. Combined with a launch configuration such as Listing 2, debugging such an Epsilon program would look as in Figure 4.

With just the launch configuration, debugging would require two interactions: starting the Gradle build, and starting the remote debugging session. To avoid this issue, VS Code allows for specifying the label of a *preLaunchTask* inside the launch configuration, to be executed before the debug session is started. The task would be defined in a separate *tasks.json* file, such as the one in Listing 4. The Epsilon VS Code extension [11] defines the *$epsilon-debug* problem matcher that detects the logging messages from the Epsilon debug server that indicate that it is waiting for connections, meaning that the debugging session can be started.

[5]Note that Maven can also reuse Ant tasks and is supported by a VS Code extension, so this is applicable to Maven builds as well. Due to time constraints, we will only demonstrate Gradle.

[6]https://eclipse.dev/epsilon/doc/articles/running-epsilon-ant-tasks-from-command-line/#gradle

Listing 4: VS Code task definition for single-click launching and debugging of an Epsilon program within a Gradle build

```
{
    "type": "gradle",
    "script": "runEOL",
    "group": "other",
    "buildFile": "${workspaceFolder}/build.gradle",
    "workspaceFolder": "${workspaceFolder}",
    "projectFolder": "${workspaceFolder}",
    "args": "--info",
    "problemMatcher": [
        "$gradle",
        "$epsilon-debug"
    ],
    "label": "epsilonDebug",
    "isBackground": true
}
```

3.5 Unifying local and remote debugging in Eclipse

After implementing remote debugging, it was noted that it had a number of advantages over the original debugging codebase which was tied to Eclipse APIs: its decoupling from UI concerns made it easier to automatically test, and it used LSP4E-based UIs that would benefit from contributions from a broader community beyond the Epsilon userbase. For those reasons, it was decided to use DAP for the original local debug configurations as well: the launch delegates were updated to use the approach in Listing 1 and delegate to LSP4E for debugging.

Given the positive experience with DAP and its support across VS Code and Eclipse, we are considering re-engineering other components of the Epsilon tooling to be based on generic implementations compliant with popular specifications. This would further reduce the effort of supporting newer IDEs in the future. The VS Code extension for Epsilon includes an LSP language server which could be reused for syntax checking and other language support details in Eclipse and other IDEs, as well as TextMate [14] grammars for syntax highlighting. TextMate grammars work on many other IDEs, such as Eclipse (using TM4E [10]), or IntelliJ IDEA.

4 Related work

Remote debugging is a common feature in general-purpose programming languages. Some examples include the Java Debug Wire Protocol (JDWP) [18], the *gdbserver* for C/C++ [1], or *pydevd* for Python [21] (which was originally a debugger for the Python support in Eclipse, and has been reused since in other IDEs such as PyCharm and VS Code).

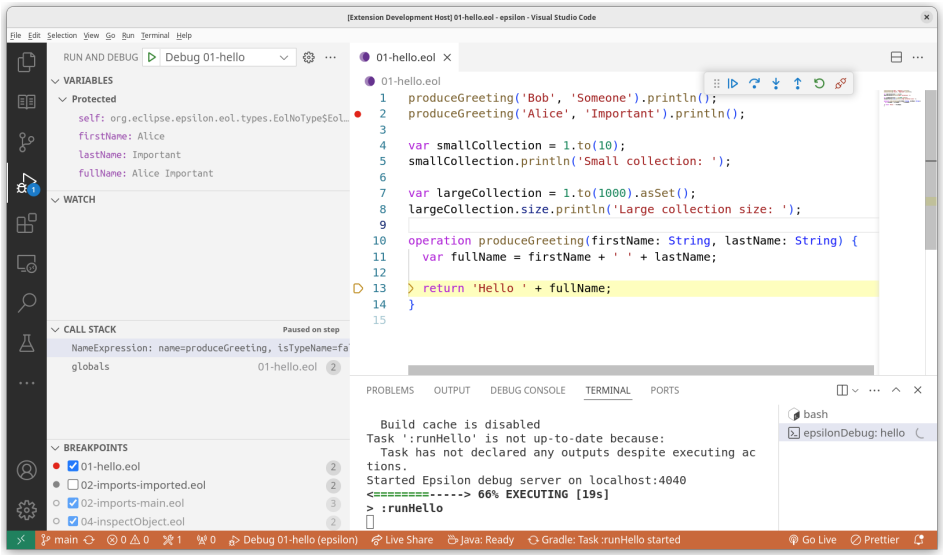

Figure 4: Screenshot of the VS Code debugger stopped at a breakpoint inside an EOL program

Remote debugging appears to be less common in domain-specific languages (including those dedicated to model management). There is a Github project providing DAP support to the Rascal metaprogramming language [20]. The Acceleo model-to-text language moved to LSP-based editors and DAP-based debugging in version 4, back in 2020 [13], but we have not found any documentation on using Acceleo outside the Eclipse IDE. We searched "lsp" through the QVTo 3.10.8 sources [3] and could not find any mentions of LSP4J or the DAP protocol. We similarly inspected the sources of Eclipse OCL 6.21.0 [5], ATL 4.10.0 [4], and VIATRA 2.8.1 [6], and could not find any mentions of LSP4J or DAP.

5 Conclusions and future work

We have presented the new remote debugging capabilities in the Eclipse Epsilon languages for its upcoming 2.6 release, describing the design used to support its languages in a maintainable way, and the user experience in two IDEs (Eclipse and VS Code) and two build systems (Ant and Gradle). We plan to continue this line of work on re-engineering Epsilon on top of common specifications in order to reduce the work involved in supporting newer IDEs, with the adoption of the Epsilon LSP language server in the VS Code extension into the main codebase, and the redesign of the current Eclipse-based editors on top of the LSP4E and TM4E projects.

Acknowledgments

This research on remote debugging of model management programs was funded by the SCHEME InnovateUK project (#10065634).

References

[1] Gary Benson. 2015. Remote debugging with GDB. https://developers.redhat.com/blog/2015/04/28/remote-debugging-with-gdb Last accessed: 2024-07-03.

[2] Justin Cooper, Alfonso De la Vega, Richard Paige, Dimitris Kolovos, Michael Bennett, Caroline Brown, Beatriz Sanchez Pina, and Horacio Hoyos Rodriguez. 2021. Model-Based Development of Engine Control Systems: Experiences and Lessons Learnt. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, Fukuoka, Japan, 308–319. https://doi.org/10.1109/MODELS50736.2021.00038

[3] Eclipse Foundation. 2023. Eclipse QVTo 3.10.8 sources. https://git.eclipse.org/c/mmt/org.eclipse.qvto.git/tag/?h=3.10.8 Last accessed: 2024-07-03.

[4] Eclipse Foundation. 2024. Eclipse ATL 4.10.0 sources. https://github.com/eclipse-atl/atl/tree/v4.10.0 Last accessed: 2024-07-03.

[5] Eclipse Foundation. 2024. Eclipse OCL 6.21.0 sources. https://git.eclipse.org/c/ocl/org.eclipse.ocl.git/tag/?h=6.21.0 Last accessed: 2024-07-03.

[6] Eclipse Foundation. 2024. Eclipse VIATRA 2.8.1 sources. https://github.com/eclipse-viatra/org.eclipse.viatra/tree/2.8.1 Last accessed: 2024-07-03.

[7] Eclipse Foundation. 2024. The Epsilon Model Connectivity Layer (EMC). https://eclipse.dev/epsilon/doc/emc/ Last accessed: 2024-07-03.

[8] Eclipse Foundation. 2024. LSP4E GitHub project. https://github.com/eclipse/lsp4e Last accessed: 2024-07-03.

[9] Eclipse Foundation. 2024. LSP4J GitHub project. https://github.com/eclipse-lsp4j/lsp4j Last accessed: 2024-07-03.

[10] Eclipse Foundation. 2024. TM4E GitHub project. https://github.com/eclipse/tm4e Last accessed: 2024-07-03.

[11] Sam Harris. 2024. GitHub project for the Eclipse Epsilon Languages Extension. https://github.com/Arkaedan/vscode-epsilon/ Last accessed: 2024-07-03.

[12] Dimitrios S. Kolovos, R.F. Paige, and Fiona Polack. 2006. The Epsilon Object Language (EOL). In *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*. Springer Berlin Heidelberg, New York, NY, USA, 128–142. https://doi.org/10.1007/11787044_11

[13] Yvan Lussaud. 2020. Acceleo 4 ever. https://www.eclipsecon.org/2020/sessions/acceleo-4-ever Last accessed: 2024-07-03.

[14] MacroMates Ltd. 2024. TextMate 1.5.1 documentation — Language Grammars. https://macromates.com/manual/en/language_grammars Last accessed: 2024-07-03.

[15] Microsoft Corporation. 2023. DAP Implementations. https://microsoft.github.io/debug-adapter-protocol/implementors/adapters/ Last accessed: 2024-07-03.

[16] Microsoft Corporation. 2024. Debug Adapter Protocol homepage. https://microsoft.github.io/debug-adapter-protocol/ Last accessed: 2024-07-02.

[17] Microsoft Corporation. 2024. Language Server Protocol homepage. https://microsoft.github.io/language-server-protocol/ Last accessed: 2024-07-02.

[18] Oracle Corporation. 2024. Java Debug Wire Protocol. https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/jdwp-spec.html Last accessed: 2024-07-03.

[19] Stack Overflow. 2023. Annual Developer Surveys. https://survey.stackoverflow.co/ Last accessed: 2024-07-03.

[20] Jurgen Vinju, Davy Landman, et al. 2024. Rascal Language Servers Github project. https://github.com/usethesource/rascal-language-servers Last accessed: 2024-07-03.

[21] Fabio Zadrozny. 2024. PyDev.Debugger Github project. https://github.com/fabioz/PyDev.Debugger Last accessed: 2024-07-03.