Protocol-Based Interactive Debugging for Domain-Specific Languages

Josselin Enet, Erwan Bousse, Massimo Tisi, Gerson Sunyé

JOT
JOURNAL OF
OBJECT TECHNOLOGY

Protocol-Based Interactive Debugging for Domain-Specific Languages

Josselin Enet, Erwan Bousse, Massimo Tisi, and Gerson Sunyé
Nantes Université, École Centrale Nantes, IMT Atlantique, CNRS, LS2N, UMR 6004, France

ABSTRACT

Interactive debuggers are established tools used by developers to understand programs and localize faults. They are equally valuable in the context of model-driven development, when working on executable behavioral models. However, development costs of interactive debuggers for Domain-Specific Languages (DSLs) can be significant. In order to mitigate these costs, several reusable DSL-agnostic debugging solutions have been proposed. We argue that the applicability of these solutions is limited by being tied to a fixed set of debugging services, a specific language engineering approach, or a particular user interface. In this paper, we present a novel approach to provide interactive debugging services for executable DSLs through a reusable generic architecture. We propose a protocol allowing a generic interactive debugger to communicate with heterogeneous DSL runtimes, both for controlling the execution and for configuring the debugger with domain-specific breakpoints. The proposed debugger can itself be controlled using a reinterpretation of the Debug Adapter Protocol (DAP), for an effortless integration in existing Integrated Development Environments (IDEs) that support it. Using a prototype implementation based on JSON-RPC and two heterogeneous DSL runtimes, we show that our approach provides an off-the-shelf reusable interactive debugger that supports meaningful domain-specific breakpoints, and that can be used with minimal effort within an IDE such as Visual Studio Code.

KEYWORDS Domain-specific languages, Debugging, Language tooling.

1. Introduction

Interactive debugging is an essential technique to understand how an executable program unfolds, whether to verify its correctness or to localize defects. It provides developers with additional ways to interact with a running program, such as an increased control over the execution, or the inspection of the runtime state. A wide range of interactive debuggers already exists for most existing General-purpose Programming Languages (GPLs), such as the GNU Debugger (Free Software Foundation 2023) (GDB), a well-established interactive debugger that supports a variety of GPLs, such as C, C++, Go, or Rust.

In the context of Model-Driven Development (MDD), interactive debuggers are very valuable tools when working on *behavioral models*. Such models are typically written using Domain-Specific Languages (DSLs), and express what the possible behaviors of the system under development are. Provided an execution semantics and an interactive debugger, a behavioral model can be executed and debugged, allowing the language user to better understand its unfolding behaviors. Yet, as of today, only few DSLs have their own full-fledged interactive debuggers. One likely reason is the important effort required to implement such a complex tool for a new DSL. Indeed, an interactive debugger must provide a wide range of services (stepping, breakpoints, inspection, etc.) that must be integrated with both the runtime of the considered DSL and a user interface (UI) giving access to said services.

One desirable solution is the definition of a *generic* interactive debugger that can be reused by new DSLs with little cost. Some approaches already follow this direction and offer some level of reusability for debugging services (Bousse et al. 2018; Wu et al. 2008; Lindeman et al. 2011; Pasquier et al.

JOT reference format:
Josselin Enet, Erwan Bousse, Massimo Tisi, and Gerson Sunyé. *Protocol-Based Interactive Debugging for Domain-Specific Languages.* Journal of Object Technology. Vol. 22, No. 2, 2023.
http://dx.doi.org/10.5381/jot.2023.22.2.xx

2022), including reusability for different DSLs. However, we observe that some key limitations are yet to be addressed. First, most approaches are *language-parametric*, i.e. , they consider the DSL as a "white box" that they introspect in order to dynamically provide adapted services. While this is a powerful way to generalize an approach, it requires strong assumptions on how supported DSLs are implemented—such as the used language engineering approach, with specific metalanguages and frameworks—and thus hinders reusability. Second, most approaches are tied to a specific UI and therefore cannot be easily integrated in existing Integrated Development Environments (IDEs). Third, reusable interactive debuggers usually aim at providing a fixed set of generic debugging services that are suitable for any DSL. They are not easily able to provide services that are specific to the domain of a particular DSL, such as domain-specific breakpoints. A solution that addresses one of these limitations may easily have a negative impact on another. For instance, considering a DSL as a "black box" make it difficult for the debugger to adapt to a domain, and providing domain-specific debugging services may prevent from reusing existing generic UIs. We therefore claim that these limitations should be addressed all at once.

We present in this paper a novel approach to provide interactive debugging services for executable DSLs through a reusable generic architecture. The main contributions are as follows:

- We introduce the Language Runtime Protocol (LRP), a protocol allowing a generic debugger to communicate with a language runtime. LRP takes inspiration from recently popular language protocols, such as LSP (Microsoft 2023b) or GLSP (Eclipse Fundation 2023). *By supporting LRP, a generic debugger can be reused for DSLs that are implemented in different language-engineering approaches, metalanguages, and frameworks*.
- We introduce the Configurable Debug Adapter Protocol (cDAP), a protocol for integrating IDEs with interactive debuggers for DSLs. The protocol cDAP is compatible with a subset of the Debug Adapter Protocol (DAP) (Microsoft 2023a), which is nowadays supported by most IDEs. We argue that DAP is structurally not fully adapted to the debugging of DSLs, hence cDAP redefines the semantics of some DAP methods and provides a minimal set of additional configuration services. *By supporting cDAP, a generic debugger for DSLs can be reused in different IDEs*.
- We introduce the concept of *breakpoint type*, and include it in both LRP and cDAP. Breakpoint types allow defining and reasoning upon the different types of breakpoint a DSL may require. A generic debugger may use the breakpoint types defined for a given DSL to configure domain-specific breakpoints accordingly. While in this paper we focus exclusively on them, domain-specific breakpoints are only one of the possible domain-specific debugging services that a DSL debugger may need to provide. The example of breakpoint types allows us to show how *our architecture enables the inclusion of domain-specific debugging services in the generic debugger without breaking reusability*.

We developed a prototype implementation of the proposed interactive debugger using TypeScript and JSON-RPC. We use this prototype to evaluate our approach with two heterogeneous DSL runtimes: State Machines—implemented using Python and ANTLR (Parr & Quong 1995)—and MiniTL—a minimal model transformation language running on a Java Virtual Machine, implemented using EMF (Steinberg et al. 2008), Xtext (Eysholdt & Behrens 2010) and Kermeta3[1]. For each considered heterogeneous DSL, results show that our approach can: (1) provide an off-the-shelf reusable interactive debugger, (2) be used with minimal effort within a standard IDE such as Visual Studio Code, (3) be used to define meaningful breakpoints types.

To evaluate our approach, we propose to answer the following research question:

- **RQ1**: Is the proposed interactive debugging approach DSL-agnostic? In other words, can it provide an interactive debugger for any executable DSL, regardless of both the domain of the DSL and of how the DSL runtime is implemented?
- **RQ2**: How much effort is required to adapt an existing DSL runtime to make it compatible with the proposed protocol?
- **RQ3**: Can the proposed interactive debugging approach be used to define and use relevant domain-specific breakpoints types for different sorts of DSLs?
- **RQ4**: Can the proposed interactive debugger be integrated with limited effort in an existing IDE that supports DAP?

The rest of the paper is structured as follows. Section 2 presents the scope of DSLs considered in the paper, standard interactive debugging services, and the concept of language protocol. Section 3 explains our contribution, which is a protocol-based approach to provide generic interactive debugging for executable DSLs. Section 4 presents the implementation of our prototype. This implementation is then used for the evaluation described in Section 5. Section 6 lists previous related work related to our research matter. Finally, Section 7 summarizes the contribution of this paper and highlights research perspectives.

2. Background

In this section, we present the scope of considered executable DSLs, the main services usually found in interactive debugging solutions, and finally, an overview of language tooling protocols.

2.1. Executable DSLs

A DSL is a usually small language specialized to a particular technical or application domain. In the context of MDD, a DSL can be used to create models that each define a particular facet of a system. A DSL commonly comprises both a syntax defining what are models conforming to the DSL, and a semantics defining the meaning of each conforming model.

Syntax The syntax of a DSL can be decomposed in an abstract syntax, defining the concepts of the DSL and their relationships, and a concrete syntax, defining how to graphically or textually represent each concept. A model conforming to the DSL has

[1] http://diverse-project.github.io/k3/

a concrete representation—e.g. text or diagrams—conforming to the concrete syntax, and this concrete representation can be translated in an abstract representation conforming to the abstract syntax. In this paper, we make no assumption on how the syntax of a DSL is defined (e.g. using a metamodel-based or a grammar-based approach), and we assume the abstract representation of a model can always be mapped to an Abstract Syntax Tree (AST) with cross-references (i.e. a node of the tree may contain an explicit reference to another node).

Execution Semantics The semantics of a DSL define some effect occurring when processing a conforming model, e.g. the generation of an artifact (e.g. code, documentation), some execution yielding a result, or the generation of a program that is executed on the fly. We call *execution semantics* a semantics defining how a model is executed. An execution semantics defines both what are the possible runtime states of a conforming model being executed, and how the runtime state of a model changes over time due to occurring *execution steps*. In this paper, we focus only on DSLs with execution semantics, which we call *executable DSLs*. While we make no assumptions on how an execution semantics is defined (e.g. using a translational or operational approach), we make the following assumptions: (1) The execution of a model can always be represented as a sequence of execution steps, each yielding a set of changes on the runtime state; (2) The runtime state can be mapped to a list of named structured elements, which may include cross-references to elements of the AST; (3) A list of input parameters can be passed when executing a model, in order to set initial values in the runtime state.

DSL Runtime We call *DSL runtime* the implementation of an executable DSL as executable software in any arbitrary form (e.g. a set of executable programs or a language server). A DSL runtime should provide all required facilities to work with a language, such as the ability to parse a model represented as a file, to analyze a model, or to execute a model to yield results. A wide range of programming languages, metalanguages, tools and frameworks—sometimes available combined in the form of a language workbench—can be used to implement a language runtime. Examples include the Eclipse Modeling Framework (Steinberg et al. 2008), ANTLR (Parr & Quong 1995), Xtext (Eysholdt & Behrens 2010), Langium[2], MPS[3], Spoofax (Kats & Visser 2010). We do not make assumptions on how a DSL runtime is implemented, as long as it satisfies our previous assumptions about the syntax and the semantics of considered DSLs.

Running Example: State Machines DSL Figure 1 shows an example of an executable DSL for State Machines, inspired from UML state machines (Object Management Group 2023). The abstract syntax of this DSL is defined as a metamodel, i.e. an object-oriented model representing the concepts of the language as classes. A state machine is composed of multiple states, one of them being referenced as the initial state. States are linked to each other or to a pseudo-state via transitions; a transition has an input and output associated to it. There are different kinds of states and pseudo-states:

- Final pseudo-states only have incoming transitions.
- Simple states have a name and can have incoming and outgoing transitions.
- Composite states have the same properties as simple states, and contain internal states, as would a state machine.

At the start of an execution, a sequence of input parameters is used to initialize the runtime state. The values stored by the runtime state are initialized as follows: *inputs* takes the value of the inputs passed as arguments, *nextConsumedInputIndex* takes the value 0, *outputs* is an empty array, and *currentState* references the initial state of the state machine. The only kind of execution step present in this language is to fire a transition. To fire a transition, the language finds a transition going out of the current state and with an input equals to the current input being consumed; this input is stored in *inputs* at the index *nextConsumedInputIndex*. Firing a transition changes the runtime state as such: *nextConsumedInputIndex* is incremented by 1, the output value produced by the transition is appended to *outputs*, and *currentState* references either the target of the transition if it is a simple state, or the initial state contained in the target composite state. The execution stops either when all inputs have been consumed, when a final pseudo-state is reached, or when no transition to be fired can be found.

Figure 2 presents a state machine model represented using the same graphical concrete syntax as UML state machines. The model is composed of four states A, B, C, and D. A is a simple state as well as the initial state of the state machine. B is a composite state, reachable through a transition from A and with transitions going back to A or to a final pseudo-state. It contains two simple states C and D, C being the initial one. A transition goes from C to D, and conversely. This model also contains instances of domain-specific breakpoints, which are further described in Section 3.

An example of execution of this state machine model can be achieved using the following input parameters: ["b", "d", "a", "b", "final"]. The first two execution steps of this execution are the firing of the transition AtoB, and the firing of the transition CtoD. When both these steps are over, the runtime state will comprise the following values:

- *inputs*: ["b", "d", "a", "b", "final"]
- *outputs*: ["AtoB", "CtoD"]
- *nextConsumedInputIndex*: 2
- *currentState*: SimpleState D

Finally, we consider that this State Machines DSL is implemented in the form of a DSL runtime running in a Python Virtual Machine. More precisely, the abstract syntax is defined using a set of manually written Python classes, the concrete syntax is managed using both ANTLR4 and some boilerplate code, and the operational semantics is written in pure Python in the form of a simple loop that fires transitions as long as inputs are available. A Python executable is able to start the execution of a state machine model stored as a text file conforming to the grammar of the DSL.

[2] https://langium.org/
[3] https://www.jetbrains.com/mps/

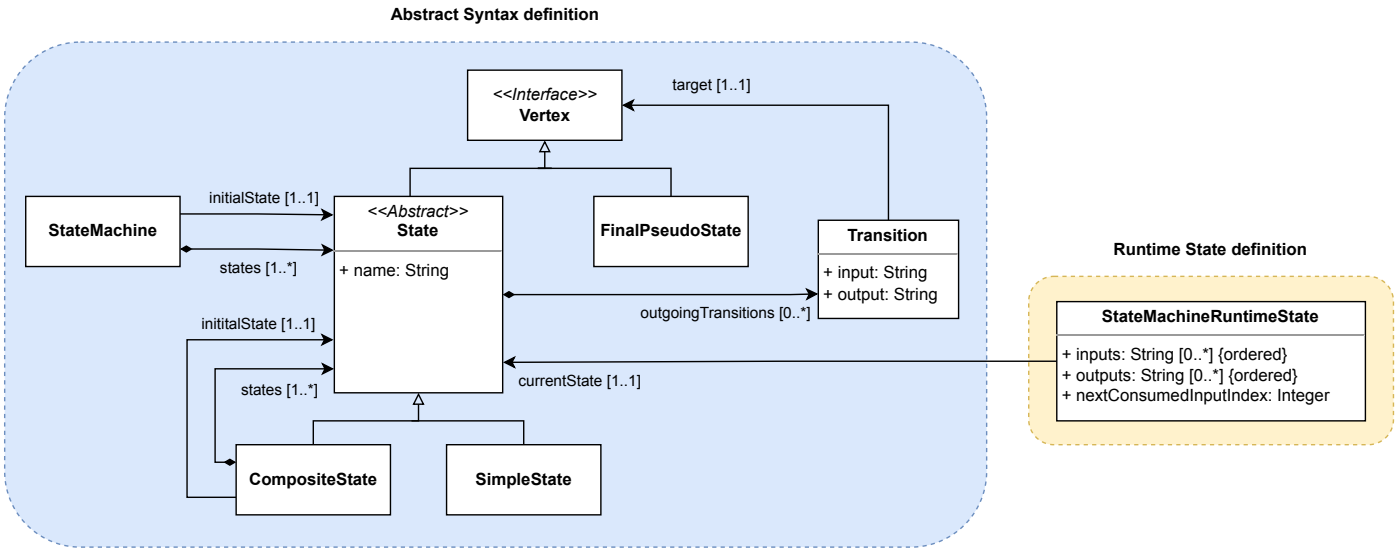

Figure 1 Abstract syntax and runtime state definitions of the considered State Machines DSL

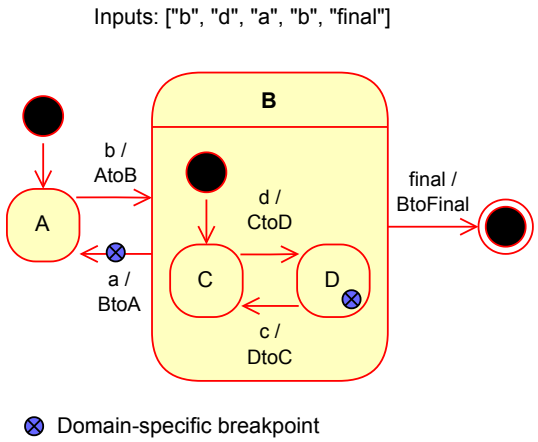

Figure 2 Example state machine model, conforming to the abstract syntax shown in Figure 1, depicted with a graphical concrete syntax.

2.2. Interactive Debugging

This section presents some standard features found in most interactive debuggers (Zeller 2023) and the scope of features we consider in this paper. We also use as a reference the Debug Adapter Protocol (DAP) (Microsoft 2023a), which is a standardized definition of these features for typical imperative GPLs.

Breakpoints Interactive debugging allows users to pause the execution of a running program using *breakpoints*. A breakpoint defines a condition on the running program, which is then evaluated at each execution step. When this condition is verified, the execution is paused, allowing the developer to observe the runtime state of the program and to use different debugging actions to control the execution, such as *stepping* operators (presented thereafter) or *resuming* the execution.

The most common and well-known type of breakpoints is called *source breakpoints*. A source breakpoint can be set to a specific position in the program. Then, before a semantic operation is executed on an AST element present at the given position, the execution pauses. When the considered language has a textual concrete syntax, source breakpoints are commonly set on a specific line and possibly column of the source file, and trigger a stop before the execution of statement or expression found in this line. Other types of breakpoints exist: *variable breakpoints* can be set on variables in runtime state, and are triggered when the value stored in the variable changes; *exception breakpoints* are triggered when a specific type of exception is thrown; and others.

Stepping Operators Once the execution is paused, a second way for a developer to control execution is through the use of *stepping operators*. These operators can be used to perform *debugging steps*, i.e. to execute only a specific portion of a program based on the current runtime state without having to manually set breakpoints. For example, a usual debugging step is to execute only the line at which the execution is paused. The most known common stepping operators are *step over* to skip the inner execution steps of the next program element (e.g. to hide the internal steps of a function call), *step into* to execute the next enclosed element of the current program element (e.g. to observe the internal steps of the body a function).

Variables View When the execution is paused, the developer can inspect the current runtime state of the program using the variable view[4]. This view presents a structured representation of the current runtime state, which the developer can browse to inspect all runtime elements. The variable view is usually structured in multiple *scopes*, such as the scope of the current function, the global scope, etc. Each scope can be unfolded to show the variables it contains. Similarly, variables with complex values (such as objects or arrays) can be folded or unfolded.

[4] We use the term *view* in a general abstract sense that includes both graphical (e.g. in an IDE) and textual (e.g. in a CLI debugger) representations of variables.

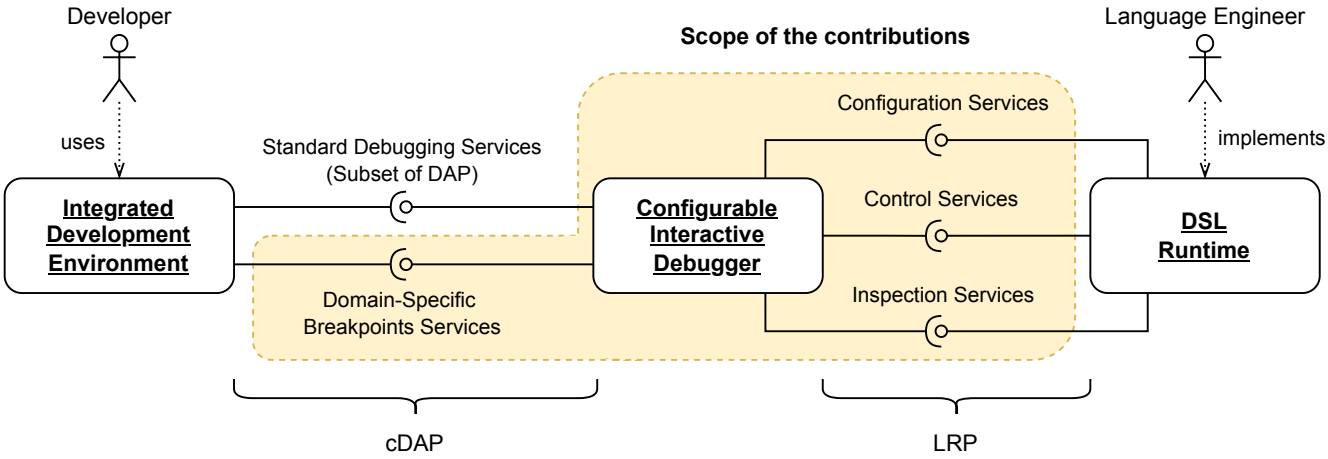

Figure 3 Overview of the proposed interactive debugging architecture

Limitations of standard interactive debugging features for DSLs We can observe that all the standard debugging features presented above are mostly all tailored for the typical features and concepts available in a standard GPL (e.g. Java, TypeScript or C#). They are however rarely adapted for DSLs that do not include GPL-oriented concepts such as statements, variables, functions or exceptions. We claim that a DSL requires *domain-specific interactive debugging services* that are fitting for the features and concepts it provides. In this paper, we propose a first example of such services in the form of *domain-specific breakpoint types*, which can be defined for any given DSL, and used within a generic interactive debugger.

2.3. Language Protocols

When providing tools or user interfaces for software languages, a difficulty lies in how the tool should communicate with the runtime of the language it is made for. A current trend in language engineering is to facilitate this communication using well-defined language interfaces (Degueule et al. 2017) and in particular language protocols (Jeanjean et al. 2021). The idea of language protocols is to consider that a language runtime should run in a *server*, which should provide services over a (usually local) network. These services can then be used by an IDE to automatically provide UIs and tools for the language, regardless of how a language runtime is implemented.

Among the most popular and successful language protocols are the Language Server Protocol (LSP) (Microsoft 2023b) and the Debug Adapter Protocol (DAP) (Microsoft 2023a). LSP is designed for the textual edition of programs, and allows a generic editing UI of an IDE (e.g. Visual Studio Code, Eclipse, IntelliJ) to rely on textual editing services directly provided by the language runtime. Similarly, DAP is designed for the debugging of textual programs, and allows a generic debugging UI of an IDE to communicate with debugging services provided by the language runtime directly.

Note that in both examples presented above (i.e. LSP and DAP), the protocol requires complex services to be defined *within* the language runtime (e.g. editing or debugging services), and their goal are only to reuse the UI related to these services. In contrast, in this paper, we choose to define a reusable generic interactive debugger that is defined *outside* the language runtime, and that communicates with the latter using a first dedicated protocol. We then use a second protocol — which is an extension to DAP — to integrate this generic interactive debugger within any DAP-compatible IDE.

3. Protocol-Based Interactive Debugging

This section presents our generic interactive debugging architecture for executable DSLs. We first give an overview of the proposed solution, then we present the protocols we define for the integration of heterogeneous DSL runtimes, and finally we describe the proposed generic interactive debugger.

3.1. Overview

Figure 3 presents the general architecture of the framework proposed in this paper. The generic debugger is contained in its own independent component. It communicates with two other components: a language runtime and a UI. Communication with language runtimes is performed through a new protocol, the Language Runtime Protocol (LRP), which can be implemented by a variety of languages and allows for customization of debugging features. In this paper, we focus on the definition of domain-specific breakpoints. Exchanges with UIs are ensured by a marginally revised version of DAP, together with some additional services for the management of domain-specific breakpoints: the joint usage of these protocols is referred to as the configurable Debug Adapter Protocol (cDAP).

In the following sections, we further describe the structure and semantics of these protocols, as well as the responsibilities of a generic debugger. We also illustrate the interaction of all these components in an end-to-end use case.

3.2. Language Runtime Protocol

In order for debuggers to communicate in a unified fashion with various language runtimes, we defined a new protocol: the Language Runtime Protocol (LRP). Through this protocol, debuggers are able to start and control the execution of a program.

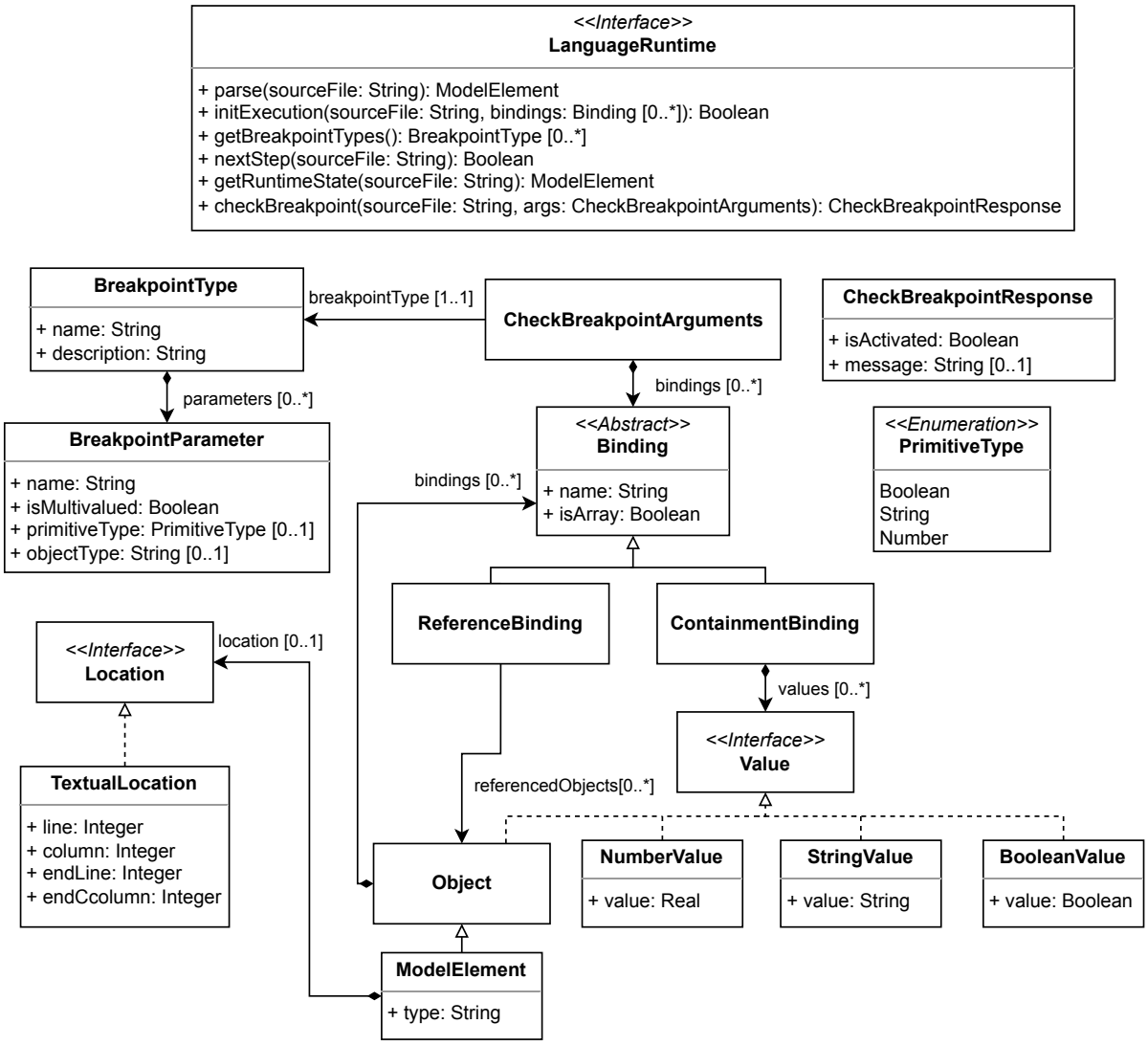

Figure 4 Class diagram of the Language Runtime Protocol (LRP)

Moreover, language runtimes can configure a debugger with language-specific breakpoints. Note that we do not make any claim about the completeness of this protocol. Our goal is simply to propose a protocol to which a debugger can map cDAP calls for the usual debugging features described in Section 2, so that language runtimes do not have to completely reimplement cDAP. Whether the services of LRP are sufficient to describe any sort of breakpoint is not a concern addressed in this paper, even though the protocol was designed with genericity in mind.

Figure 4 describes the structure of LRP. The *LanguageRuntime* interface describes the services that a language runtime must provide in the context of this protocol. As mentioned in Figure 3, three groups of services can be identified in LRP, which we define shortly after.

Figure 2 presents a state machine with domain-specific breakpoints, represented by blue crossed circles attached to states or transitions. On this particular instance, a breakpoint is attached to the state D and another breakpoint to the transition from B to A. We rely on this example to better describe each service.

Configuration Services The *parse* service requests the language runtime to parse a given source file. After calling this service, the language runtime has access to the AST associated with the source file. Calling this service also returns a model to the debugger; this model represents the AST, but under a form that is more intelligible by end-users of the debugger. This means that the internal AST stored by the language runtime and the AST received by the debugger (i.e. , the one manipulated by end-users during debugging) can differ. For instance, a language runtime for the State Machine DSL can internally manipulate an AST in the form of a complex transition matrix. Directly giving access to this representation to the end-user might render debugging difficult. Instead, the language runtime can present the AST under a much more readable form, such as a model with concrete State and Transition objects. Each element of the AST has a type; the debugger does not need to know the relationship between the types of the language, therefore it is sufficient to store this type as a string in each element. Finally, elements to which a domain-specific breakpoint can be assigned come with

a location; it represents a range in the source file where putting a source breakpoint will translate to assigning a domain-specific breakpoint to the corresponding element. This means that while the ASTs manipulated by the debugger and language runtime can be structurally different, the breakpoint types defined by the language runtime are constrained by the AST passed to the debugger. This is because locations for domain-specific breakpoints are attached to elements of the AST manipulated by the debugger. Please also note that since we propose in this paper to reuse DAP for the communication with UIs and since DAP is tailored for textual languages, LRP only currently supports locations in textual source files. However, it is perfectly conceivable to add support for other kinds of locations in LRP, such as ones in graphical representations.

The *initExecution* service performs the initialization of the runtime state of a program, based on the AST and possible additional arguments passed in the service call. This results in the language runtime storing a runtime state corresponding to the state before the execution of any execution step. In addition, this service returns a boolean, conveying whether an execution step can be performed by the language runtime. On the running example, calling this operation results in the language runtime initializing its runtime state as follows:

- *inputs*: ["b", "d", "a", "b", "final"]
- *outputs*: []
- *nextConsumedInputIndex*: 0
- *currentState*: SimpleState A

The boolean *true* is returned since the transition from A to B can be fired when consuming the first input.

The *getBreakPointTypes* service retrieves a list of all breakpoint types made available by the language runtime. For convenience, each breakpoint type comes with a human-readable name and description. A breakpoint type can have multiple parameters; each parameter has a name and a type. This type is either a primitive type (boolean, string or number) or an object type, which is simply represented as a string. A parameter can be single- or multivalued. For example, a breakpoint type that breaks when a state is reached takes one parameter: a single value of type State as defined by the language runtime.

Control Services The *nextStep* service asks the language runtime to perform the next execution step for a given program, and returns a boolean representing whether another execution step can be performed (same as *initExecution*). Calling this service on the running example will set the runtime state as follows:

- *inputs*: ["b", "d", "a", "b", "final"]
- *outputs*: ["AtoB", "CtoD", "BtoA"]
- *nextConsumedInputIndex*: 3
- *currentState*: SimpleState A

The boolean *true* is returned since the transition from A to B can be fired when consuming the fourth input.

Inspection Services The *getRuntimeState* service returns a representation of the current runtime state stored by the language runtime. As for the AST, the representation of the runtime state

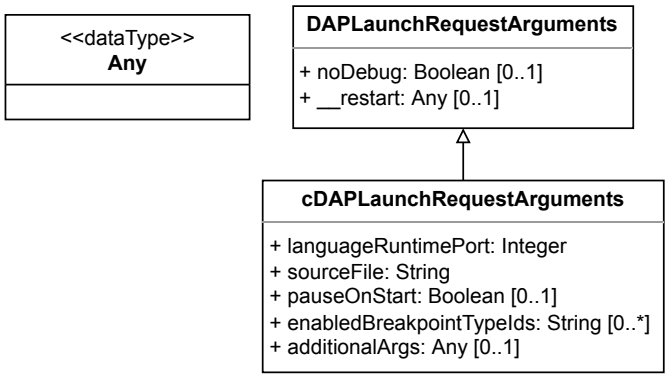

Figure 5 Class diagram of the altered arguments of the *launch* DAP service

manipulated by the language runtime and the debugger can differ. Elements of the runtime state have the same structure as elements of the AST; however, elements of the runtime state can not only reference other runtime state elements, but also AST elements. The runtime state of the running example is already presented in Section 2. Note that the *currentState* reference points to a State contained in the AST.

The *checkBreakpoint* service passes a breakpoint type along with additional arguments to the language runtime; these additional arguments can be model elements or any other arbitrary value. The language runtime checks whether the predicate associated to the given breakpoint type is verified on the next execution step when applied with the given arguments, and returns a corresponding boolean together with a message destined to the user if the breakpoint was activated. Let's consider the breakpoint type mentioned during the presentation of the *getBreakpointTypes* service. To check whether an instance of this breakpoint is activated, this breakpoint type must be passed to *checkBreakpoint* along with a State from the AST. In the situation depicted in Figure 2, this service will signal an activated breakpoint just before the second input is consumed, since this input triggers a transition targeting state D.

3.3. DAP for Domain-Specific Debugging

Trying to combine DAP, which is geared towards imperative GPLs, with the use of domain-specific breakpoint types is a conflicting endeavor. Still, we argue that interfacing with DAP is important since it is a widely adopted protocol to integrate debuggers with existing UIs[5]. The main difficulty in using this protocol is that it defines data structures that are strongly coupled to a subset of executable languages. For instance, different types of breakpoints are already defined in the protocol: source breakpoints, variable breakpoints, function breakpoints, instructions breakpoints, etc. This makes it complex to use domain-specific breakpoint types in conjunction with this protocol. To address this issue, we propose to slightly alter the semantics of DAP while keeping the same service signature. We also add new services for enabling domain-specific breakpoint types.

[5] https://microsoft.github.io/debug-adapter-protocol/implementors/tools/

DAP Services	Original Semantics	Altered Semantics
initialize	Configures the debugger with the client's capabilities, and vice-versa.	Same as original.
disconnect	Disconnects the program being debugged from the debug adapter and shuts down the debug adapter. If the debug session was started through the *launch* service, the program being debugged must also be terminated.	Always terminates the program being debugged, but keeps the debugger running.
launch	Starts the execution of a given program, with or without debugging.	Same as original, but contains additional arguments described in Section 3.3.
threads	Retrieves a list of all threads.	Retrieves a mock, unique thread.
stackTrace	Retrieves the stack trace of the current execution state for a specified thread. A stack trace is composed of a range of stack frames.	Retrieves a stack trace containing a mock, unique stack frame.
scopes	Retrieves the scopes for a specified stack frame. A scope stores a reference that can be used to retrieve its variables through the *variables* service.	Retrieves two mock scopes: one for the AST, and one for the runtime state.
variables	Retrieves the child variables for a specified variable reference.	Same as original.
next	Performs a step in the current granularity for the specified thread. Can prevent other threads from resuming if the corresponding capability is supported by the debug adapter.	Performs an execution step (no thread notion).
stepIn	Performs a step into (presented in Section 2) for the specified thread. Can prevent other threads from resuming if the corresponding capability is supported by the debug adapter.	Performs an execution step (no thread notion).
stepOut	Performs a step out (presented in Section 2) for the specified thread. Can prevent other threads from resuming if the corresponding capability is supported by the debug adapter.	Performs an execution step (no thread notion).
continue	Resumes the execution of all threads of a program. Can prevent other threads from resuming if the corresponding capability is supported by the debug adapter.	Resumes the execution of the program (no thread notion).
setBreakpoints	For a given source file, sets multiple source breakpoints and clears all previous source breakpoints.	Same as original, except that the semantics of source breakpoints are different; they change depending both on the syntax element a source breakpoint is associated to, and the enabled breakpoint types during runtime.

Table 1 List of reused DAP services in cDAP

Altered DAP Table 1 shows the list of services from DAP reused in cDAP, along with their original and altered semantics. This alteration introduces room for domain-specificness; first, we can use the existing source breakpoints to implement domain-specific breakpoints. Source breakpoints can be attached to a precise location in a source file (i.e. a line and a column); as a result, this location can be used to pinpoint a specific element of the syntax. As mentioned in the presentation of LRP, language runtimes expose a service to check whether a breakpoint is verified. Since we can derive elements from a location in the concrete syntax, a debugger can map source breakpoints set through DAP to calls to the *checkBreakpoint* service of LRP. For instance, the State Machine DSL can expose a breakpoint type that is triggered when a given state is about to be reached. By putting a source breakpoint at the location corresponding to a state, the debugger is able to infer the state targeted by the domain-specific breakpoint. A limitation of relying on source breakpoints is that resulting domain-specific breakpoints can only be parameterized by exactly one element of the abstract syntax. As an example, it is impossible to define a breakpoint type that would break when a given state is reached for the n^{th} time, where n is an additional parameter that can be set by the

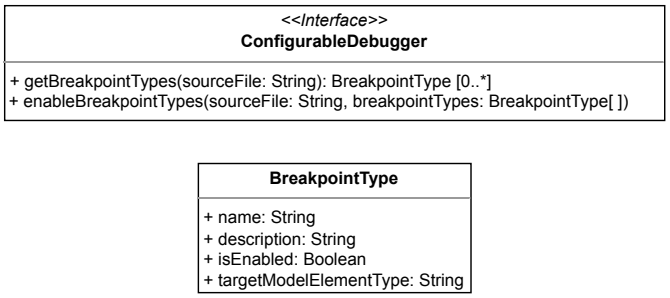

Figure 6 Class diagram of the domain-specific breakpoints services of cDAP

end-user. It is also impossible to define breakpoints over values exclusively stored in the runtime state, such as outputs.

To start the execution of a program, the *launch* service of DAP can be reused, but additional arguments must be introduced to support domain-specific debugging. Figure 5 describes these additional arguments. Arguments present in the original *launch* service of DAP, represented in the class *DAPLaunchRequestArguments*, are still usable in our altered version of the service. The altered version is presented in the class *cDAPLaunchRequestArguments*. There are two mandatory arguments: the integer *languageRuntimePort* stores the local port at which the language runtime related to the program to launch is listening to; the program to run is identified in the string *sourceFile*. The boolean *pauseOnStart* specifies whether the debugger should pause the execution after the initialization of the runtime state, but before executing any step. The array of string *enabledBreakpointTypeIds* contains the identifiers of all breakpoint types that should be enabled at the start of the execution. Finally, *additionalArgs* contains an arbitrary data structure that will be forwarded to the language runtime to initialize the execution of the program through the *initExecution* LRP service.

In the same spirit, we can reuse the services of DAP that are used to fill the variables view: *threads*, *stackTrace*, *scopes*, and *variables*. These services again make some assumptions about features present in the language, such as the use of threads or the presence of a stack trace. We attribute a mock value to these specific features and only use the part that contributes to showing the content of the runtime state in the variables view. A drawback of this approach is that languages that do manipulate these features, such as thread and stack trace, can not currently profit from the specific support that DAP offers.

The stepping support of DAP remains unchanged in cDAP: domain-specific stepping operators are out of the scope of this paper. This means that any step performed by the end-user will translate to an execution step in the language runtime.

Additional Services During debugging, end-users will want to choose what breakpoint types they wish to use. We address this case with new services in the debugger interface in addition to usual DAP services.

Figure 6 show these additional services. *getBreakpointTypes* asks the debugger to list all the domain-specific breakpoint types available for a given source file. Each breakpoint type is identified by a unique string, and targets exactly one type of element present in the AST (also identified by a unique string). For convenience, each breakpoint type also comes with a human-readable name and description. A breakpoint type also carries a boolean which determines whether it is currently enabled. The State Machine DSL can for instance provide breakpoints that are triggered when a state is reached, a transition is fired, an input is consumed or an output is produced.

enableBreakpointTypes asks the debugger, for a given source file, to enable all the breakpoint types passed as parameters. All previsouly enabled breakpoint types that are not passed as parameters are disabled. For instance, the State Machine DSL can provide three breakpoint types: one triggered when a state is reached, another when a state is exited and another when a transition is fired. It is possible to enable any combination of these breakpoint types, including no breakpoint types at all.

Algorithm 1 : *continue* service of the configurable debugger

Input:

$file$: the source file of the program

$proxy$: the proxy of the language runtime

b_{inst} : the breakpoint instances on the running program

b_{act} : the breakpoint instances already activated since the last execution step

```
begin
    isExecutionDone ← false
    while ¬isExecutionDone do
        foreach b ∈ b_inst do
            if b ∈ b_act then continue
            target ← getTargetModelElement(b)
            foreach b_type ∈ getEnabledBreakpointTypes(target.type)
            do
                args ← getCheckBreakpointArguments(b, b_type)
                response ← proxy.checkBreakpoint(file, args)
                if response.isActivated then
                    b_act ← b_act ∪ b
                    rs ← proxy.getRuntimeState(file)
                    updateRuntimeState(rs)
                    notifyPauseToUI(response.message)
                    return
        b_act ← {}
        isExecutionDone ← proxy.nextStep(file)
    notifyTerminationToUI()
```

Debugger In the present approach, the debugger acts as the bridge between the UI and the language runtime. This point alone distinguishes our approach from DAP, where debugger and language implementation are not necessarily separated. The main responsibility of a debugger in our framework is then to map cDAP calls coming from the UI to internal operations in the debugger. Some of these internal operations will in turn require calls to LRP services provided by language runtimes. Algorithm 1 depicts the pseudo-code describing how the debugger

handles a cDAP call to the *continue* service.

While the execution is not done, the debugger will execute two tasks in sequence: first, check if any breakpoint is activated and second, request the language runtime to perform a step. The debugger goes through all breakpoints stored in b_{inst}; if an instance was already activated during the current execution step, it is ignored. Then, for each enabled breakpoint type associated to the type of model element targeted by the instance, the debugger uses $proxy$ to call the *checkBreakpoint* LRP service. If the response to this request is the activation of the breakpoint, then the instance is added to the list of activated breakpoints b_{act} for the current execution step, the runtime state stored in the debugger is updated, and a notification is sent to the UI. Once all breakpoints have been checked for a given step, b_{act} is cleared and a request to the *nextStep* LRP service is sent through $proxy$. If the response is that the execution is over, a notification is sent to the UI. Otherwise, the main loop is entered again.

Note that the debugger has the responsibility of tracking elements to which domain-specific breakpoints are associated; in this regard, language runtimes only have to be able to check whether a breakpoint is verified or not, on demand of the debugger through the appropriate LRP service. However, preserving breakpoints on a source file between executions is still the responsibility of the UI, as is normally the case with DAP.

3.4. End-to-End Use Case

Figure 7 depicts a scenario in which a user debugs the State Machine previously presented in Figure 2.

The exchange begins with sequence of calls to *initialize*, *launch* and *setBreakpoints*; this is standard DAP communication. The inputs to be consumed by the state machine are passed in the *launch* request, as well as the breakpoint types enabled at the start of execution: we consider that a unique breakpoint type is enabled, which breaks when a specific state is reached. As a result of this request, the debugger initiates a sequence of calls to the *parse*, *initExecution* and *getBreakpointTypes* LRP services presented by the State Machines language runtime. At the end of this sequence, the language runtime has initialized a runtime state for the program and is ready to execute the first. The debugger has access to a representation of the AST of the program, knows which breakpoint types are provided by the language runtime and can map source breakpoints to these domain-specific breakpoints.

Since the UI was storing a previously assigned source breakpoint, a *setBreakpoints* request is sent along the *launch* request. Once the debugger is updated on which breakpoint types are available, it is able to map the source breakpoint to a domain-specific breakpoint and store it. Then, the UI is notified about the success of the setting of the breakpoint.

At this point, the main loop of the debugger—similar to the one presented in Algorithm 1—can begin. The breakpoints are successively checked before executing the first step by calling the *checkBreakpoint* LRP service. Here, the breakpoint assigned to the transition is not activated, since no breakpoint type that takes a transition as argument is enabled. The breakpoint on state D is not activated either since the target of the first transition is C. Consequently, the debugger requests the language runtime to execute the next step through the *nextStep* LRP service; after this step, the execution is not yet over since more inputs can be consumed. The debugger checks again all the breakpoints; this time, the breakpoint on state D is activated because D is the target of the second transition, and this breakpoint has not been already activated for this execution step. As a result, the debugger adds this breakpoint to the set of breakpoints activated for this execution step. It also updates its representation of the runtime state by calling the *getRuntimeState* LRP service, so that it is able to appropriately respond to requests of the UI to populate the variables view. Finally, it notifies the UI from the pause in the execution and suspends operations until new requests are received from the UI.

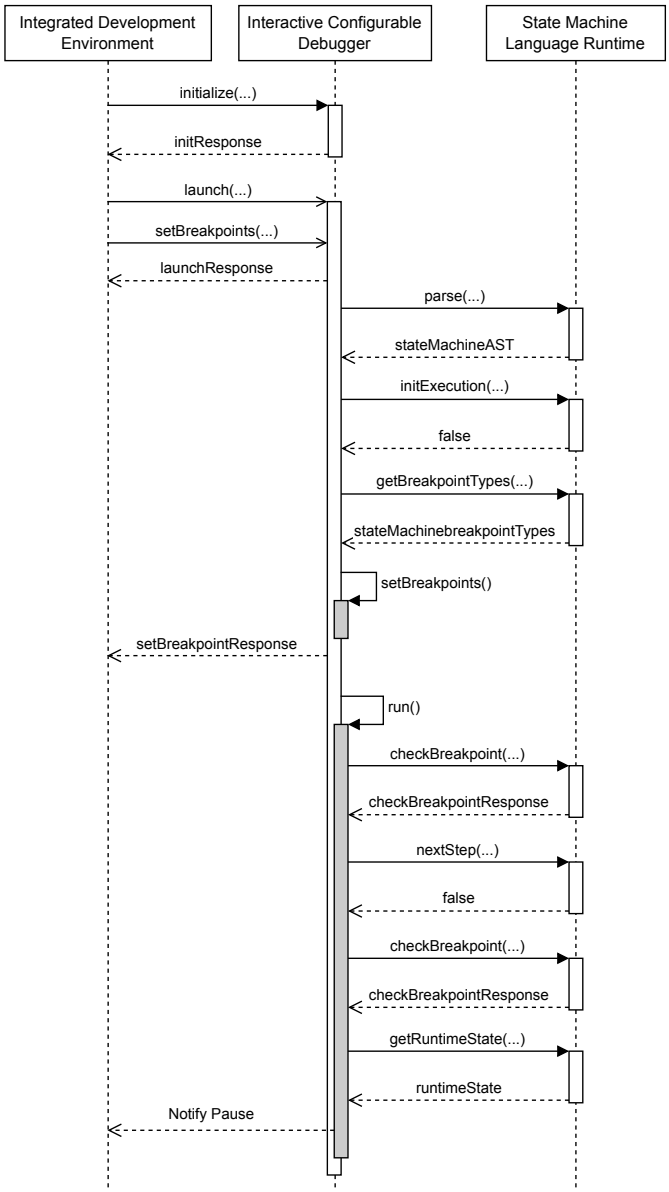

Figure 7 Execution scenario with a configurable debugger and a State Machines language runtime implementing LRP

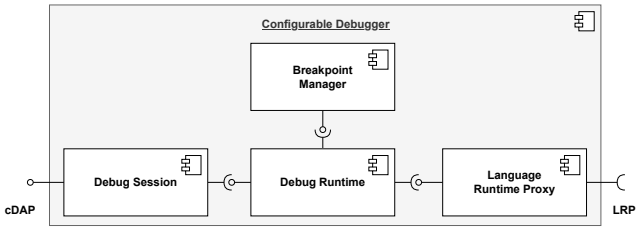

Figure 8 Architecture of the prototype debugger

4. Implementation

This section presents the implementation of two components that follow our proposed approach: a configurable debugger and a Visual Studio Code extension adding support for cDAP.

4.1. Configurable Debugger

Our prototype of a configurable debugger[6] is implemented in TypeScript. Its overall architecture is presented in Figure 8. It communicates with language runtimes through a JSON-RPC version of LRP. A JSON Schema specification is available[7] for both LRP and cDAP.

The debugger itself can be decomposed in four main components. * The *Language Runtime Proxy* ensures the communication with the language runtime through LRP. In the case of this prototype, we work with a JSON-RPC version of our protocol, without closing the possibility to use other network protocols. This proxy is used by the *Debug Runtime*; this component keeps track of all the relevant information about the debugging of a given program. It relies on the *Breakpoint Manager* for all things related to breakpoints; this includes tracking which breakpoints are present in the running program, as well as checking their activation before each step. Finally, the *Debug Session* is responsible for the communication with the UI through cDAP. The role of this component is to translate cDAP requests to calls to the Debug Runtime. For the sake of simplicity, our current implementation of the Debug Session can only handle the debugging of one program at a time.

4.2. Visual Studio Code Extension

Visual Studio Code[8] is a lightweight IDE distributed by Microsoft. It supports communication with debuggers through DAP. We integrate our debugger in Visual Studio Code through an extension developed in TypeScript. This extension does not have to re-implement DAP communication, which is already generically handled by Visual Studio Code. Its goals are:

- To declare languages that implement LRP services, and can therefore be used in conjunction with our debugger. The authors have no yet identified a way to dynamically discover which languages are compatible with our debugger, so this declaration has to be manually performed in the extension.
- To implement the additional services over DAP related to domain-specific breakpoints. Together with DAP, this forms the cDAP protocol used to communicate with our configurable debugger.

This extension does not yet support the automated launch of either the debugger or language runtime. Rather, these components are expected to be already running. An explanation of the launch process and the manipulation of domain-specific breakpoints is given in the repository of the prototype.

[6] https://github.com/NaoMod/Protocol-Based-Interactive-Debugging-for-DSLs-Prototype
[7] https://github.com/NaoMod/Configurable-Debugger-Protocols-Specification
[8] https://code.visualstudio.com

5. Evaluation

This section describes how we evaluated our contributions according to the research questions listed in Section 1.

5.1. Experimental setup

We detail below our experimental setup.

Considered DSL runtimes and models Table 2 summarizes the two DSL runtimes considered for our evaluation. The first is the Python-based runtime of the State Machines DSL already presented in Section 2.1. The second is a Java-based runtime of a minimal model transformation language called MiniTL.

Loosely inspired from ATL (Jouault et al. 2008), MiniTL can be used to define a model transformation from an input Ecore metamodel to an output Ecore metamodel. A MiniTL transformation is composed of a set of rules each with an input pattern and an output pattern. Each input pattern defines how to match an object conforming to the declared input metamodel, while each output pattern defines what element conforming to the output metamodel to create. The runtime of MiniTL is implemented with the Eclipse Modeling Framework (EMF), using Ecore for the abstract syntax, Xtext for the concrete syntax, and Kermeta for the operational semantics. The MiniTL runtime therefore runs on a JVM.

For each executable DSL, we considered two different small executable models, each in the form of a text file conforming to the grammar of the DSL.

Considered breakpoint types We aim to define different breakpoint types for each considered DSL runtime. For the State Machines DSL, we wish to have two types of breakpoints for *states*: one for when a state is *about to be reached*, and one for when a state is *about to be exited*. We also consider a breakpoint type for when a transition is *about to be fired*. For MiniTL, we consider two breakpoint types: one when a transformation rule is *about to be applied to all matching elements*, and one when a feature of an element of the output model is *about to be assigned a value*.

Considered IDE To demonstrate that our approach can be integrated in any IDE that supports DAP, we focused on one IDE currently popular among developers: Visual Studio Code.

Procedure and metrics Our evaluation aims to assess how our prototype implementation, previously presented in Section 4, can be used to debug models conforming to the two DSL runtimes presented above. Our evaluation comprises three phases: DSL runtime extension, IDE tooling, and model debugging.

Language	Language Runtime Implementation		Considered Breakpoint Types
	Technological Space	Approach	
State Machines	Python	ANTLR4	• State Reached • State Exited • Transition Fired
MiniTL	Java	XText + EMF + Kermeta3	• Rule Applied To All Elements • Value Assigned to Feature

Table 2 Summary of the executable DSLs considered in the experimental setup

In the *DSL runtime extension* phase, we extend each considered DSL runtime to implement the services expected by our proposed Language Runtime Protocol (LRP), including the services required for each *breakpoint type* considered for the DSL. As required by our prototype, these services must follow the JSON-RPC standard, and must translate the queries they receive into actual calls to the internal implementation of the DSL. We measure the effort required to build this integration layer with the amount of lines of code that must be written. This phase contributes to answering RQ1, RQ2 and RQ3.

In the *IDE tooling* phase, we integrate the proposed generic interactive debugger in the considered IDE, i.e. Visual Studio Code. This requires a Visual Studio Code extension with (1) the required boilerplate code for the pure DAP-based communication with the debugger, and (2) the required code to both communicate with the debugger through our extension of DAP for domain-specific breakpoints, and to provide UI elements to interact with these services. Again, we measure the effort required to build this integration layer with the amount of lines of code that must be written. This phase aims to answer RQ4.

Finally, in the *model debugging* phase, we execute each considered model of each considered DSL runtime using the complete end-to-end setup: the considered IDE, itself integrated with the generic interactive debugger, itself integrated with the DSL runtimes. With each execution, we create one breakpoint per considered breakpoint type, and we use the *continue* service to reach each breakpoint. This phase contributes to answering RQ1 and RQ3.

5.2. Results

We discuss below the results obtained through the evaluation phases for each considered research question.

RQ1 In the runtime extension phase, we implemented the LRP interface for both the State Machines and MiniTL language runtimes. These runtimes implement DSL with distinct domains and use different implementation technologies. The LRP protocol itself can then be considered DSL-agnostic.

During the model debugging phase, we executed the models considered for each DSL. For all models, we were able to create a breakpoint of each considered type. The creation of breakpoints was realized in a unified fashion, by placing a source breakpoint on the source file of each model. The execution stopped correctly when the condition corresponding to each breakpoint was met. We therefore consider that our approach can provide interactive debugging in a DSL-agnostic way.

RQ2 In the runtime extension phase, existing runtimes were modified to support LRP services. The exact effort required to implement LRP is difficult to quantify, since some code might be embedded into the existing runtime code. For the State Machines DSL, 589 lines of code (LoC) were necessary; we estimate that half of this code could be reused in other language runtimes implemented in Python. Regarding MiniTL, 1116 LoC were required to integrate support for LRP. Again, we suggest that half of this code could be reused for other runtimes depending on Java. While the implementation effort to support LRP is important we argue that it is still easier to achieve for language engineers than to re-implement a complete debugger. In addition, this approach becomes increasingly interesting as the capabilities of the debugger evolve; the implementation effort for language engineers stays the same.

RQ3 In the runtime extension phase, we extended each language runtime with the capacity to check domain-specific breakpoints through the *checkBreakpoint* LRP service. It is possible to call this service to verify the condition of a breakpoint during the execution of a program by the language runtime. As such, LRP is sufficient for language runtimes to define domain-specific breakpoints.

In the model debugging phase, models were executed for each considered DSL runtime. Source breakpoints were put directly on the source files of the models, and were then translated to domain-specific breakpoints. The semantics of these breakpoints could be switched during runtime thanks to the available breakpoint types. Hence, we consider that our approach allows the definition and usage of domain-specific breakpoints for heterogeneous DSLs. Note that other usual debugging features, such as stepping operators and variables view, were working but were not configured in a domain-specific way. More specifically, using any of the stepping operators resulted in the execution of a single execution step.

RQ4 During the IDE tooling phase, we developed an extension in TypeScript for VSCode as to integrate our configurable debugger. 147 LoC were necessary for the implementation, 132 of which were dedicated to the extension of DAP for domain-

specific breakpoints. Since VSCode already supports communication through DAP, no additional code is required to interact with the debugger through standard DAP services. Thus, we consider that the implementation effort required to integrate our debugger in IDEs that already support DAP is reasonable.

5.3. Limitations

Our evaluation does not currently include an assessment of the usability of our approach by real developers. This would require finding a set of developers that use an executable DSL on a regular basis and would beneficiate from debugging features.

The evaluation would also profit from a comparison with other DSL-agnostic debugging approaches, such as ones listed in Section 6. However, this analysis is made complicated by the fact that none of these approaches has exactly the same scope as ours; some focus on advanced debugging features, others don't consider integration with multiple UIs, etc. Also, the variety in the technological spaces of the different solutions adds another layer of complexity for this comparison.

In general, the measure of lines of code is not very representative of the complexity of the implementation. However, we consider that the use cases present in our evaluation are simple enough to not warrant the use of other metrics. If more complex use cases are explored in the future, then the addition of supplementary measures may be interesting.

6. Related Work

Several language engineering workbenches provide solutions for debugging, but offer limited options for defining domain-specific debugging services. For instance Spoofax (Kats & Visser 2010) provides a DSL for the declarative definition of debuggers (Lindeman et al. 2011). Debug events exposed by the language must be mapped to a predefined set of event classes. These existing classes might not be sufficient to express the variety of events that can be produced by each DSL.

The Eclipse GEMOC Studio debugger (Bousse et al. 2018) is a debugger with advanced capabilities, such as reverse execution, that is available for DSLs at a reduced implementation cost. It exists inside the Eclipse GEMOC Studio (The GEMOC Initiative 2023), an environment for the development and use of executable DSLs. The Eclipse GEMOC Studio debugger is composed of a standalone component, to which *engines* can be connected. These engines are responsible for hiding approach-specific details during the execution of a program, and present a standardized interface through which the GEMOC debugger can interact with the running program. Therefore, it is possible to make new approaches work with the debugger without having to completely re-implement a debugger. Another notable aspect of this debugger is that breakpoints can be associated to elements of the graphical syntax, providing some level of customization. However, the breakpoint type associated to each graphical element is automatically deduced from the language implementation, with no means to configure them. The main limitation of this framework is that it can only be exploited by languages implemented in one of the approaches available in the Eclipse GEMOC Studio. While new approaches can be added to the Eclipse GEMOC Studio, an underlying requirement is that they must all be based on the Eclipse Modeling Framework (Steinberg et al. 2008) (EMF).

The Moldable Debugger (Chiş et al. 2014) proposes a framework to define domain-specific operations and views for internal DSLs implemented inside object-oriented host languages. Some components of the debugger, handling low-level communication with the host language, can be reused to build new domain-specific debugging constructs. Still, the debugger implementation is strongly coupled to the host language: for instance, the prototype implementation of the Moldable Debugger is written in Pharo, and can only deal with DSLs hosted in this language. Domain-specific breakpoints can be described over primitive debugging predicates available on all DSLs. However, these predicates can only be used through GPLs—in the case of the prototype implementation, Pharo—and no infrastructure is provided to easily define these breakpoints.

Multiverse debugging is a debugging method focused on providing operations that ease the debugging of non-deterministic languages (Pasquier et al. 2022). The G∀min∃ debugger interfaces itself with various languages through a formally-defined interface: the Semantic Language Interface. This interface presents an *evaluate* service, which takes an expression in an arbitrary formalism and returns the result of its evaluation over the runtime state of the program. This approach is useful to let end-users define their own breakpoint types; however, it forces language runtimes to implement the semantics of an expression formalism in addition to its own semantics. Another limitation of this work is its compatibility with existing UIs; the G∀min∃ debugger is currently implemented in the AnimUML environment in an ad-hoc manner.

Leroy et al. (Leroy et al. 2020) describe *behavioral interfaces*, i.e. means to interact with a program during its execution. These interfaces are tailored for DSLs with discrete-event operational semantics, i.e. DSLs that can react to incoming stimuli. In contrast, our approach considers that DSLs receive no external inputs passed the execution initialization.

Besnard et al. (Besnard et al. 2018) define a unified semantics interface for the simulation, formal verification and execution of UML models. In comparison, our approach thrives to be applicable to languages implemented in heterogeneous ways and is geared towards execution and debugging.

7. Conclusion

Reducing the development cost of interactive debuggers for Domain-Specific Languages (DSLs) is a necessary undertaking to bring DSLs on par with General-purpose Programming Languages (GPLs). We proposed in this paper a reusable generic architecture to reduce the effort required to build such tools, and we demonstrated with two DSLs that with only reasonable integration effort, the resulting interactive debugger can both be compatible with heterogeneous DSLs, and allow the definition of domain-specific debugging services (i.e. breakpoint types).

We see different complementary research directions to continue this work. The evaluation can be extended both with more IDEs (e.g. Eclipse or IntelliJ), and with a larger selection of DSL

runtimes, in order to cover more domains and more language engineering techniques. The proposed Language Runtime Protocol could be extended to cover concurrent execution semantics (i.e. with non-determinism and parallel composite steps) (Zschaler et al. 2023). More configuration domain-specific debugging services could be provided, such as composite steps or runtime state with different scopes. The flexibility of the domain-specific breakpoint types available to the user could be increased by not relying on cDAP for the communication between the UI and the debugger. Lastly, the development of the integration layer of a DSL runtime could be facilitated, e.g. by defining helpers for each considered language engineering approach (in the same spirit as *execution engines* of the GEMOC Studio (Bousse et al. 2016)).

References

Besnard, V., Brun, M., Jouault, F., Teodorov, C., & Dhaussy, P. (2018). Unified LTL verification and embedded execution of UML models. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (pp. 112–122). doi: 10.1145/3239372.3239395

Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., & Combemale, B. (2016, October). Execution framework of the GEMOC studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN international conference on software language engineering*. ACM. doi: 10.1145/2997364.2997384

Bousse, E., Leroy, D., Combemale, B., Wimmer, M., & Baudry, B. (2018, March). Omniscient debugging for executable DSLs. *Journal of Systems and Software*, *137*, 261–288. doi: 10.1016/j.jss.2017.11.025

Chiş, A., Gîrba, T., & Nierstrasz, O. (2014). The moldable debugger: A framework for developing domain-specific debuggers. In *International Conference on Software Language Engineering* (pp. 102–121). Springer. doi: 10.1007/978-3-319-11245-9_6

Degueule, T., Combemale, B., & Jézéquel, J.-M. (2017). On language interfaces. In *Present and ulterior software engineering* (pp. 65–75). Springer International Publishing. doi: 10.1007/978-3-319-67425-4_5

Eclipse Fundation. (2023). *GLSP*. https://www.eclipse.org/glsp/.

Eysholdt, M., & Behrens, H. (2010). Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (pp. 307–309). doi: 10.1145/1869542.1869625

Free Software Foundation. (2023). *GDB: The GNU Project Debugger*. https://www.sourceware.org/gdb/.

Jeanjean, P., Combemale, B., & Barais, O. (2021, February). IDE as code: Reifying language protocols as first-class citizens. In *14th innovations in software engineering conference (formerly known as india software engineering conference)*. ACM. doi: 10.1145/3452383.3452406

Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008, June). ATL: A model transformation tool. *Science of Computer Programming*, *72*(1-2), 31–39. doi: 10.1016/j.scico.2007.08.002

Kats, L. C., & Visser, E. (2010, October). 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* (pp. 444–463). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1869459.1869497

Leroy, D., Bousse, E., Wimmer, M., Mayerhofer, T., Combemale, B., & Schwinger, W. (2020, July). Behavioral interfaces for executable DSLs. *Software and Systems Modeling*, *19*(4), 1015–1043. doi: 10.1007/s10270-020-00798-2

Lindeman, R. T., Kats, L. C., & Visser, E. (2011). Declaratively defining domain-specific language debuggers. *ACM SIGPLAN Notices*, *47*(3), 127–136. doi: 10.1145/2189751.2047885

Microsoft. (2023a). *Official page for Debug Adapter Protocol.* https://microsoft.github.io/debug-adapter-protocol/.

Microsoft. (2023b). *Official page for Language Server Protocol.* https://microsoft.github.io/language-server-protocol/.

Object Management Group. (2023). *Unified Modeling Language Specification.* https://www.omg.org/spec/UML/.

Parr, T. J., & Quong, R. W. (1995). ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience*, *25*(7), 789–810. doi: 10.1002/spe.4380250705

Pasquier, M., Teodorov, C., Jouault, F., Brun, M., Roux, L. L., & Lagadec, L. (2022, October). Practical multiverse debugging through user-defined reductions: Application to UML models. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems* (pp. 87–97). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3550355.3552447

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

The GEMOC Initiative. (2023). *GEMOC Studio*. https://gemoc.org/studio.html.

Wu, H., Gray, J., & Mernik, M. (2008, August). Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience*, *38*(10), 1073–1103. doi: 10.1002/spe.863

Zeller, A. (2023). How debuggers work. In *The debugging book*. CISPA Helmholtz Center for Information Security. https://www.debuggingbook.org/html/Debugger.html. Retrieved 2023-01-06 17:58:51+01:00, from https://www.debuggingbook.org/html/Debugger.html (Retrieved 2023-01-06 17:58:51+01:00)

Zschaler, S., Bousse, E., Deantoni, J., & Combemale, B. (2023, January). A generic framework for representing and analyzing model concurrency. *Software and Systems Modeling*. doi: 10.1007/s10270-022-01073-2

About the authors

Josselin Enet is a PhD student in software engineering at Nantes University (France). His thesis revolves around providing reusable language tooling for DSLs, regardless of meta-languages and domain. You can contact the author at josselin.enet@ls2n.fr.

Erwan Bousse is an Associate Professor at Nantes University (France). He obtained his PhD in France in 2015 at the University of Rennes 1 for his work on execution traces and omniscient debugging of executable models. His current research interests include Software Language Engineering (SLE), Model-Driven Engineering (MDE), Domain-Specific Languages (DSLs), model execution and simulation, and the debugging and testing of models. You can contact the author at erwan.bousse@ls2n.fr or visit https://bousse-e.univ-nantes.io/.

Massimo Tisi is an associate professor at the Institut Mines-Telecom Atlantique (IMT Atlantique, Nantes, France), and deputy leader of the NaoMod team, LS2N (UMR CNRS 6004). Since 2019 he coordinates the Lowcomote Marie Curie European Training Network. His research interests revolve around software and system modeling, domain-specific languages and applied logic. He contributes to the design of the ATL model-transformation language and investigates the application of deductive verification techniques to model-driven engineering. You can contact the author at massimo.tisi@ls2n.fr.

Gerson Sunye is an associate professor at the Nantes University (France) in the domain of software engineering and distributed architectures and the head of the Nantes Software Modeling Group. He received the PhD degree in Computer Science from the University of Paris 6, France, in 1999. From 1999 to 2001 he was a postdoctoral researcher at the IRISA Computer Science laboratory. He has 4 years of industry experience in software development. He received his Habilitation in 2015. He is the author of several papers in international conferences and journals in software engineering. His research interests include software testing, design patterns and large-scale distributed systems. You can contact the author at gerson.sunye@ls2n.fr or visit https://sunye-g.univ-nantes.io/.