Example-Based Live Programming for Everyone

Building Language-Agnostic Tools for Live Programming with LSP and GraalVM

Fabio Niephaus
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
fabio.niephaus@hpi.uni-potsdam.de

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Jakob Edding
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
jakob.edding@student.hpi.de

Jonas Hering
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
jonas.hering@student.hpi.de

Bastian König
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
bastian.koenig@student.hpi.de

Kolya Opahle
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
kolya.opahle@student.hpi.de

Nico Scordialo
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
nico.scordialo@student.hpi.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

Abstract

Our community has explored various approaches to improve the programming experience. Although many of them, such as Example-Based Live Programming (ELP), have shown to be effective, they are still not widespread in conventional programming environments. A reason for that is the effort required to provide sophisticated tools that rely on run-time information. To target multiple language ecosystems, it is often necessary to implement the same concepts, but for different languages and runtimes. Two emerging technologies present an opportunity to reduce this effort significantly: the Language Server Protocol (LSP) and language implementation frameworks such as GraalVM's Truffle. In this paper, we show how an ELP system can be built in a language-agnostic way by leveraging these two technologies. Based on our approach, we implemented the Babylonian Programming system, an ELP system that has previously only been implemented for exploratory ecosystems. Our system, on the other hand, brings ELP for all languages supported by the GraalVM to Visual Studio Code (VS Code). Moreover, we outline what a language-agnostic infrastructure needs to provide and how the LSP could be extended to support ELP also independently from programming environments. Further, we demonstrate how our approach enables the use of ELP in the context of polyglot programming. We illustrate the consequences of our approach by discussing its advantages and limitations and by comparing the features of our system to other ELP systems. Moreover, we give an outlook of how tools that rely on run-time information could be built in the future. This in turn might motivate future tool builders and researchers to consider implementing more tools in a language-agnostic way from the start to make them available to a broader audience.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**; *Software maintenance tools*; *Runtime environments.*

Keywords: Live Programming, Exploratory Programming, Language Server Protocol, GraalVM, Truffle, Visual Studio Code

ACM Reference Format:
Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '20), November 18–20, 2020, Virtual, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3426428.3426919

Onward! '20, November 18–20, 2020, Virtual, USA

ACM ISBN 978-1-4503-8178-9/20/11.
https://doi.org/10.1145/3426428.3426919

1 Introduction

Our community has created various approaches to improve the programming experience through Exploratory and Live Programming [26], in particular through tools making use of run-time information such as Whyline or Example-Based Live Programming (ELP) [1, 13]. Nevertheless, many of these tools are not widespread, in particular not in conventional programming environments. This is often not the result of a conceptual limitation, as many of these tools do not depend on specific programming languages, programming environments, or execution environments. We argue that part of the problem is the effort required to provide a tool for different ecosystems, as this means re-implementing the user interface (UI) of the tool in the corresponding programming environments and the necessary instrumentation in the corresponding execution environments. While some adaptations for specific environments will always be required, the core mechanisms of a tool may not differ much between implementations and could be reused.

Two recent trends provide an opportunity that may enable tool builders to support various environments and languages, while only implementing the core mechanisms of their tools once: the Language Server Protocol (LSP) and the Truffle instrumentation framework. The LSP decouples Integrated Development Environments (IDEs) from concrete programming languages, so that programmers are free to choose what kind of IDE they use for writing code in a specific language [18]. The Truffle instrumentation framework [30], on the other hand, enables language-agnostic implementations of program instrumentation infrastructures. This framework is part of the Truffle language implementation framework used to implement all languages for the GraalVM [33]. More importantly, Truffle already provides an LSP server based on its instrumentation framework [28].

An example for sophisticated tool support that may be implemented based on these technologies is ELP. ELP narrows the gap between static source code and the dynamic behavior of a program. While programmers write code, an ELP system uses user-provided examples to invoke annotated functions, and to provide fine-grained, live feedback on the resulting program behavior. Currently, most existing implementations of ELP systems are based on highly specialized programming systems and adapted execution environments. For widespread adoption, however, ELP needs to be available in conventional, general-purpose IDEs and should work with unmodified execution environments.

As a result of implementing tools in a language-agnostic way, programmers can always use their preferred set of tools to develop in different languages. This makes the programming experience more consistent [20]. Moreover, such tools can also be used in the context of polyglot programming, where programmers can build their applications with multiple language. The main premise of polyglot programming is to allow programmers to use the best language, library, framework, or tool for the job and thus fosters software reuse. On the other hand, this also means that they have to deal with additional cognitive overhead, for example distinguishing between different language semantics. An ELP system can help programmers to better understand such differences when building polyglot applications.

In this paper, we show how a new infrastructure can provide a complex programming tool with a live feedback loop for multiple languages. In particular, we describe and evaluate how an ELP system can be built in a language-agnostic way on top of the LSP and the Truffle instrumentation framework. On the one hand, our approach makes ELP usable uniformly across languages and development environments. On the other hand, it also enables ELP for polyglot programming, where programmers not only have to deal with cognitive challenges of one, but many languages at the same time. To demonstrate our approach, we implement the Babylonian Programming system [23, 25], an ELP system that has previously only been implemented for exploratory ecosystems, on top of Truffle's LSP implementation and the its extension for Visual Studio Code (VS Code).

Contributions:

- An approach for extending the Language Server Protocol with a live feedback loop to support ELP systems,
- A language-agnostic implementation strategy for a Babylonian Programming system, and
- A discussion of to what extent ELP can be provided in a language-agnostic and environment-agnostic way, based on a prototypical implementation of said approach and strategy.

In the remainder of this paper, we give a short introduction to ELP and provide an overview of current implementation strategies to illustrate how these result in specialized implementations in Section 2. In Section 3, we introduce LSP and the Truffle instrumentation framework. Based on this, we present our general approach for a language-agnostic ELP implementation in Section 4. In Section 5, we describe technical details of the implementation. To illustrate the consequences of our approach, we demonstrate the achieved programming experience with two walkthroughs in Section 6 and provide a discussion of the feasibility of providing ELP in a language-agnostic and environment-agnostic way in Section 7. Finally, we discuss related work in Section 8 and state our conclusions in Section 9.

2 Example-Based Live Programming: Features and Implementations

To illustrate our approach, we will re-implement an existing ELP tool using a language-agnostic infrastructure. The tool

to be implemented is called the "Babylonian Programming[1] system" [23, 25].

To demonstrate some typical features of ELP systems, we will first provide a short overview of the features of the system (see Figure 1). Further, we give an overview of existing implementations and their implementation approaches to illustrate the effort currently required to provide ELP.

2.1 Features of ELP Environments

In the following we will illustrate features of ELP systems by explaining some of the features of the Babylonian Programming system. Most of these features are also found in other ELP systems.

As the name suggests, a central feature of ELP systems is that they allow programmers to express examples and associate them with executable elements of the programming language (see 2, 6 in Figure 1). In the original Babylonian Programming system, programmers can define multiple, named examples for each function or method in JavaScript (JS). These examples might be created ad-hoc by the current programmers or might have been created by earlier programmers and left in the source code for documentation purposes.

Programmers can then activate examples to get live feedback on the program behavior. To get feedback on an expression, programmers can attach a *probe* to the expression [16] (see 4, 7 in Figure 1). The probe will show the results of all evaluations of the expression during the execution of all activated examples. Whenever programmers change code or the examples, the examples are re-executed and the probes are updated accordingly. Probes can provide feedback on simple value objects, as well as structured objects. By placing probes in different modules, programmers can see how the execution of the example involves other parts of the system.

As functions are seldomly executed in an empty context, the Babylonian Programming system allows programmers to specify the context in which the example is to be executed. One way to specify the context are *instance templates* which define instances of classes which can be re-used in examples (see 1 in Figure 1). Another way to specify the context are *replacements* which allow programmers to replace a selected expression with another expression during the execution of an example (see 5 in Figure 1). Through replacements programmers can, for example, replace a query to a database with a constant expression.

Further, in order to support programmers in checking their hypothesis about the dynamic behavior of the program, the Babylonian Programming system allows programmers to express assertions regarding the example executions. Essentially, they are probes with an additional assertion expression which is evaluated for each recorded value.

Finally, to navigate the underlying trace, the Babylonian Programming system provides graphical sliders for loops and function calls (see 3 in Figure 1). Using these, programmers can select specific iterations or function calls to limit the displayed values of probes. To ease the navigation of source code involved in the example execution, the Babylonian Programming system also grays out code that was not executed.

2.2 Implementation Strategies of ELP Systems

The implementation of an ELP system involves the programming environment as well as the run-time instrumentation. Depending on the extensibility of the programming environment and the availability of run-time instrumentation infrastructure, the ELP implementation often results in a specialized or even completely new environment. In the following, we illustrate the different implementation strategies and their consequences, for example to which degree they result in programming environment-specific implementations or how they influence the development setup.

2.2.1 Programming Environment.

On the side of the programming environment, implementers have to provide UI components for the features described above and integrate them with the infrastructure for tracing example executions.

Specialized Source Code Editor. The most common approach found in the environments listed in Table 1 is to implement a specialized code editor. While many of them are based on code editors with basic features, they generally do not integrate into a larger environment. For example, the Babylonian Programming implementations and the Seymour editor are based on commonly used code editors [10, 23, 25].

At the same time, creating a specialized source code editor gives implementers a lot of freedom to adapt the detailed user interactions within the editor. For example, the live literals editor is able to update literals in the source code to provide feedback and the editor shown in the *Inventing on Principle* demonstration is specialized for the domain of rendering [31, 32].

As these editors have been created for ELP, they are optimized for the corresponding workflows. Consequently, it may be difficult to reuse the resulting editors directly in larger environments which may use the source code editor to provide a variety of workflows.

Plug-ins. Many contemporary IDEs already provide reusable UI components, language-agnostic tooling infrastructure, such as generic interfaces for code highlighting or outlines, and a plug-in system. Some implementations make use of these to implement ELP. For example, the original example-centric environment was implemented as a plug-in

[1] The name refers to the observation that during the Babylonian period algorithms were already described using examples integrated into the abstract description of a procedure [12].

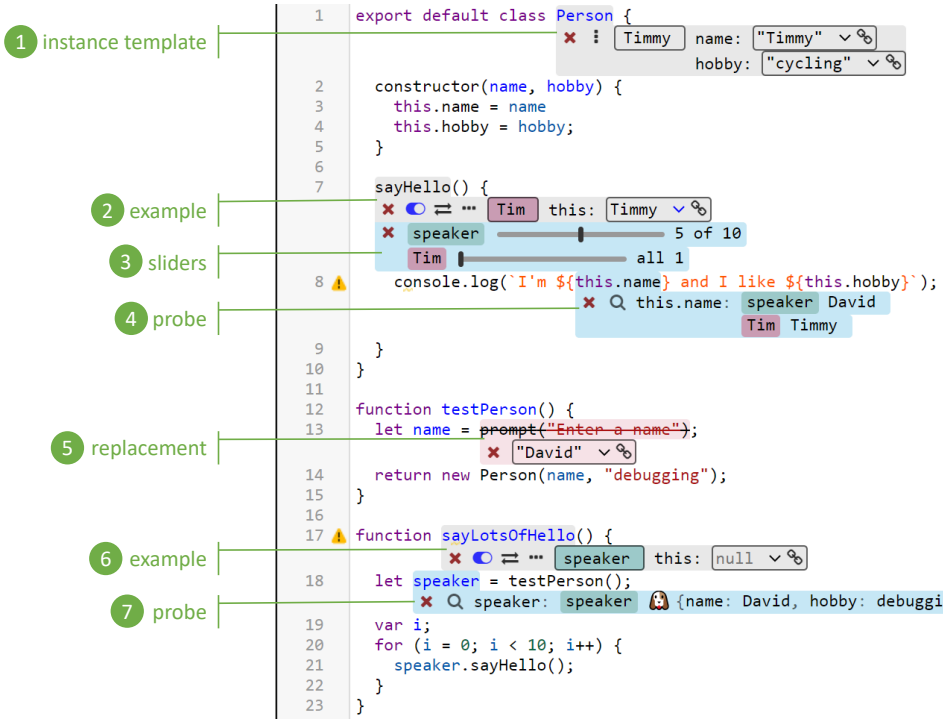


Figure 1. The editor in the Babylonian/JS system showing an active example with the name "Tim" (2) annotated to the method `sayHello()` and an active example named "speaker" (6) attached to the function `sayLotsOfHello()`. The example "Tim" uses the instance template named "Timmy" (1) to create a receiver for the method call. One probe (4) shows results for both examples. The other probe (7) shows results for the "speaker" example for a complex object. The replacement (5) substitutes the request for user action with a fixed value. The sliders (3) allow users to navigate the different calls to `sayHello()`. For the "speaker" example the fifth call out of ten is currently selected with the slider, the execution of the "Tim" example only contains one call. [23, 25]

for Eclipse and the Shiranui approach was implemented as an Emacs plug-in [1, 7].

Through reusing the existing tooling infrastructure of an IDE, implementers could provide a language-agnostic implementation of ELP. So far, however, the existing implementations all provide language-specific plug-ins.

Specialized Environment. Another way to implement ELP is to create an environment that provides infrastructure for run-time feedback. For example, the LightTable environment provides a UI element called *watches*, which are similar to probes [6]. They provide an interface that can then be implemented for particular languages, for example Clojure or Python, through plug-ins.

While this approach requires a lot of effort, it provides an infrastructure for ELP that can be reused across languages supported in that environment.

2.2.2 Instrumentation. One the side of the execution environment and the language implementation, implementers have to provide means to trace the example executions. As we are mostly interested in the user experience for programmers working with ELP, we only distinguish between implementation strategies which require programmers to use a specialized execution environment and approaches which instrument the source code before it is passed to the execution environment.

Modified Execution Environments. Several ELP implementations make use of an execution environment which is modified to trace the execution. The example-centric approach used a modified Java Virtual Machine (JVM) which yields trace information for every executed expression in combination with on-demand instrumentation [1]. Shiranui comes with a new, so far limited, programming language whose interpreter traces the evaluation of all expressions [7].

Using a modified execution environment has several advantages. First of all, instrumentation can be applied at a fine-grained level without altering the original control flow. Second, a virtual machine (VM)-level instrumentation might decrease the performance impact of tracing. However, using

a modified execution environment complicates the development setup and introduces the risk of relying on outdated or inconsistent behavior of the language implementation.

Instrumentation of Source Code. An approach which does not require an instrumented execution environment is to use an extended compilation process which introduces the instrumentation before the execution.

For example, both existing implementations of Babylonian Programming systems represent probes as comments in source code. During the compilation process, these special comments are used to instrument expressions through rewriting source code, for example by using the Babel framework in Babylonian/JS [23, 25]. LightTable stores watched ranges of source code in external structures and also adds instrumentation code before the evaluation [6].

The major advantage of this approach is that users do not have to switch to a specialized execution environment. This in turn simplifies the development setup and reduces the risk of relying on inconsistent language behavior between execution environments. At the same time, this approach can be difficult to implement, as instrumenting arbitrary expressions through source code transformations correctly is challenging. Further, tracing without VM-level support might increase the time to execute an example.

In the live literals editor, probes are ordinary function calls and thus programmers can manually instrument source code. While this circumvents the problem of correct automatic instrumentation, it also intertwines instrumentation code with program code.

3 Technology Background: Language-Agnostic Infrastructure

In this section, we introduce the technologies that enable a more generic implementation of the Babylonian Programming system that is independent from the programming language and programming environment.

3.1 Language Server Protocol

The Language Server Protocol [18] is an actively maintained, open protocol by Microsoft designed to enhance code editors and IDEs with language-specific features such as code completion, goto definitions, and various code annotations. It is based on the client-server architecture and uses JSON-RPC [9] for the communication between a tool and a *language server*. Therefore, it decouples development tools from programming languages, which in turn allows programmers to use their preferred set of tools across multiple languages.

3.2 GraalVM and Truffle

GraalVM [33] is a high-performance, polyglot VM based on the JVM and supports several different language implementations such as JavaScript, Python, and Smalltalk [19]. The Graal just-in-time compiler is written in Java and designed to run and optimize Abstract Syntax Tree (AST) interpreters. These AST interpreters must be implemented in Truffle, GraalVM's language implementation framework. This approach makes it straightforward to run polyglot applications: ASTs of different languages can be mixed and combined, which the Graal compiler can then optimize and execute. Languages can communicate with each other through the `InteropLibrary` Application Programming Interface (API), Truffle's language interoperability protocol.

Since tooling is essential when it comes to the programming experience, the Truffle framework provides numerous common development tools, such as profilers and a debugger, for both language implementers and language users. Most of these tools are built in a language-agnostic way. For supporting them, language implementers must provide implementations for both Truffle's instrumentation and language interoperability APIs. Based on these APIs, a language-agnostic implementation of the LSP for GraalVM languages was built [28] and introduced with the release of GraalVM 20.0.0.

4 Approach

The goal of this work is to demonstrate that an ELP system can be built in a language-agnostic way. For this, we leverage the LSP and extend it with a live feedback loop. Also, the approach builds on top of the instrumentation and interoperability APIs of a language implementation framework. As a result, our approach enhances conventional code editors and IDEs with ELP features independently from the programming language and thus also enables ELP for polyglot programming. In the following, we provide an overview of how this can be achieved.

Building a Live Feedback Loop on Top of the LSP. The LSP defines a messaging protocol between a language server and an LSP client, typically an extension integrated into a code editor or IDE. It has built-in support for notifications informing the server about common file operations, such as `didOpen`, `didChange`, and `didSave`. Being able to monitor source code files for changes is an important precondition for supporting live programming features. In addition, the LSP supports other notifications that a server can use to trigger specific events in a client. The specification allows custom notification messages that are ignored if not understood by a client. An LSP client, on the other hand, has access to the code editor it is integrated into, because the editor is responsible for creating appropriate UI components for incoming messages from a language server. Consequently, a live feedback loop can be integrated into the LSP by introducing appropriate messages to request the instrumented execution of code, for example every time a file is changed, and to display collected run-time information through the client in code editors.

Table 1. Overview of implementation strategies for programming environments and execution environments for ELP.

ELP Approach	Programming Environment	Instrumentation
Example-centric [1]	Eclipse plug-in	modified, tracing JVM
Live Literals [31]	specialized code editor	manual instrumentation
Shiranui [7]	Emacs plug-in	new language, tracing execution environment
Inventing on Principle [32]	specialized code editor	unknown
Seymour [10]	specialized code editor	unknown
LightTable [6]	complete environment	code rewriting
Babylonian/JS [23]	specialized code editor	code rewriting
Babylonian/S [25]	specialized code editor	code rewriting
Projection Boxes [15]	VCS extension	unknown
Brackets live coding [14]	Brackets plug-in	modified, tracing Node.js

Implementing ELP using Truffle. At the core of ELP is the ability to add examples in some form to source code. Such *exemplified code* must be executed by an appropriate runtime environment. We can implement the execution of example invocations in a language-agnostic manner, as GraalVM's interoperability API allows the execution of code from different languages in a uniform way. Moreover, GraalVM supports a sophisticated sandboxing mechanism, which is useful when dealing with intermediate and possibly incorrect versions of code under development. This mechanism allows us to set timeouts and various resource restrictions. User-defined examples can therefore be executed in such sandboxes.

More importantly, it must be possible to collect run-time information based on injected probes, assertions, and replacements to support different features of the Babylonian Programming system. The Truffle framework allows the implementation of language-agnostic instruments on the AST level. An advantage of Truffle ASTs is that they can provide lots of additional information. Each node, for example, can provide a source location, which makes it possible to map from an AST back into source code. Using Truffle's instrumentation infrastructure, additional code can be executed before and after AST nodes. This way return values can be captured on expression level, which enables probing and assertions. Furthermore, the execution of a node can be avoided by forcing a specific return value within the routine (`onEnter()`) running before a node is executed. This mechanism can be used for implementing replacements. The instrumentation framework is hence suited for collecting run-time information required for the Babylonian Programming system.

Sharing Exemplified Code Between Programmers. Examples are meant to support programmers in writing and documenting code but are negligible once a program is shipped. Furthermore, programmers often collaborate in teams through a version control system. In order to persist example and probe definitions in the code and share them with other programmers, we propose to embed these definitions in code comments. Such comments are often used for documentation purposes. Examples can also function as a documentation mechanism, probes and assertions can highlight important points in a program. Besides, most programming languages, version control systems, code editors, IDEs, and the LSP are based on files. Therefore, representing examples and probes as textual comments is more appropriate than representing them as an external artifact. Tools could provide interactive user interfaces to hide and control our definitions in code comments.

Stakeholders and Responsibilities. To illustrate why our approach enables the implementation of an ELP system that is agnostic to languages and IDEs, we describe what needs to be done to support a new language or development environment. For this, we make the following assumptions with regard to the language implementation framework and the LSP. As a prerequisite, the language framework of the runtime ecosystem needs to support language-agnostic, instrumented execution of source code. Further, the LSP specification must include an appropriate notification for displaying run-time information in the editor. This is required to make our approach fully environment-agnostic. Otherwise, it is limited to LSP clients that understand our protocol extension. For more client-side flexibility, a dedicated message to explicitly request instrumented execution of code could be added to the protocol as well. In the case of TruffleLSP, this means:

To add support for a new language, its implementers would need to implement both, the instrumentation and the interoperability protocols of the Truffle framework. If this is the case, any IDE that has an LSP client with support for the proposed LSP extension would be able to provide our Babylonian Programming features for that new language.

To add support for a new IDE, the maintainers of its LSP client — usually the maintainers of the IDE or of an IDE

extension providing an LSP client — are responsible for implementing support for our proposed LSP extension. The LSP client must be able to request the execution of exemplified code from the LSP server and to accept incoming notifications for displaying run-time information. If this is the case, the LSP client would support our Babylonian Programming system for any programming language that implements the required protocols of the language framework.

5 Implementation

This section describes how we have implemented a Babylonian Programming system based on our approach. We have extended GraalVM's TruffleLSP as well as its extension for VS Code and introduced a lightweight Domain-specific Language (DSL) for persisting examples, probes, and assertions. The code is based on GraalVM 20.0.0 and publicly available on GitHub[2].

Architectural Overview. Figure 2 gives an overview of the system's architecture: On the server side, Truffle provides the instrumentation framework and is used to implement all GraalVM languages as well as the TruffleLSP integration. To the latter, we have added a Babylonian instrument, which is the key component for evaluating and instrumenting exemplified code. On the client side, we are building on top of the LSP client that comes with GraalVM's extension for VS Code.

In addition, Figure 2 depicts how the core feedback loop functions: 1. When the user opens or changes a file, the client sends a corresponding event to the TruffleLSP. This component then uses our Babylonian instrument for collecting run-time information. The information is sent via `setDecoration` notification messages to the LSP client. Finally, the LSP client adds code decorations through VS Code's `setDecorations` API call.

Extending the TruffleLSP. TruffleLSP heavily uses Truffle's instrumentation framework. For example, it comes with a `SourceCodeEvaluator` infrastructure which is used to perform dynamic code coverage analyses. We extended the `SourceCodeEvaluator` with the ability to evaluate code for a given example. For this, we introduced a new `ExecutionEventNodeFactory`. Using a `SourceSectionFilter`, we define a filter for AST nodes tagged with the `StatementTag` to ensure our execution event nodes are only triggered by statements. In the `onReturnValue()` hook, we can then access the results for each statement and accumulate them if needed for an example, probe, or assertion. If examples are found in a file, its content is executed with our instrument. During the execution, TruffleLSP sends corresponding decoration notifications to the client asynchronously every 500ms until all example were executed. In case an example needs more time to run, a special notification is sent to inform the user that the execution of the example is in progress by displaying an ellipsis as placeholder results. Moreover, if an assertion or probe contains an expression, it is evaluated in the context of the current execution state similar to interactive code execution in the console of a debugger. Timeouts and other run-time errors are propagated to the client, so that users can see and better understand problems in their code. Since the TruffleLSP was introduced as a preview with GraalVM 20.0.0, however, it is not fully refined yet. We discuss some of its current limitations in Section 7.

Listing 1. Exemplified code using our lightweight DSL.

```
// <Example :name="hot" inputValue=95 />
function getTemperatureText(inputValue) {
 // <Probe :example="hot" />
 return `${inputValue}°F equals
          ${toCelsius(inputValue)}°C`
}
// <Example :name="cold" fahrenheit=32 />
function toCelsius(fahrenheit) {
 // <Probe :expression="fahrenheit - 32" />
 // <Assertion :example="cold" :expected=0 />
 return (fahrenheit - 32) * 5/9
}
```

Extending the GraalVM's Extension for VS Code. The key modification to GraalVM's VS Code extension is to add support for the newly introduced decoration notifications. Every time a file is changed, it is sent to the LSP backend for instrumented execution, which in turn will respond with appropriate decoration notifications, as explained in the previously paragraph. The extension distinguishes between example, probe, and assertion annotations. Examples are decorated with the return value of the function they are declared for. Probes are decorated with a string representation of the value returned as part of the line following the probe definition. Assertions work in the same way but compare a result against an expected value. Instead of a string, we use cross and check marks to encode the results of an assertion.

Our implementation supports multiple examples. To help programmers identify the right probe values and assertions for their examples, we prepend each decoration with an emoji for the corresponding example. Changing the name of an example also changes the emoji. This ensures the prefixes of decorations are consistent and short.

A DSL for Persisting Examples. To persist example, probe, and assertion definitions, we use an XML-like format as a DSL intended to be used as part of code comments. We chose an XML-like format as it is convenient to be parsed and as we assume that the syntax is familiar to many programmers, so that they can learn and memorize it quickly. Listing 1 shows an example of how this DSL can be used. Examples must have a defined name and provide valid input values using XML attributes. They also support an optional

[2] https://github.com/hpi-swa-lab/graal/tree/onward20-paper

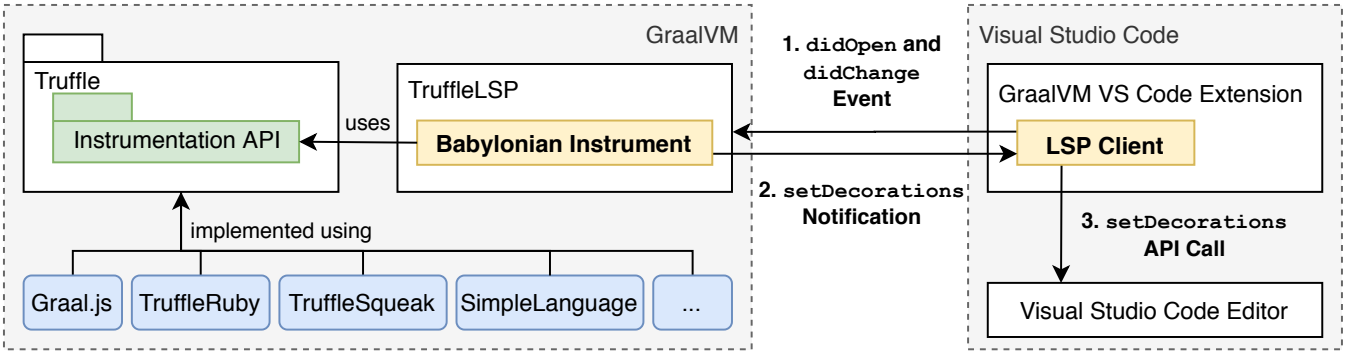


Figure 2. Architectural overview. All GraalVM languages (blue) are implemented in Truffle. The TruffleLSP uses its instrumentation framework (green). We introduced a new Babylonian instrument within the TruffleLSP and extended the LSP client (both yellow). The numbered items depict the live feedback loop from the moment the user opens or modifies a file until feedback is provided in VS Code.

`:probe-mode` attribute, which can be set to either *default*, *all*, or *off*. By *default*, only return values and explicit probes are shown. The *all* mode shows probes for all statements, similar to how debuggers, such as the Chrome debugger, show values after each line of the code. Probing can be turned *off* without having to remove an example. Probe definitions support an optional `:example` attribute. This is also the case for assertion definitions. Additionally, they require an expected value or an expression that should evaluate to true.

Furthermore, we leverage Truffle and its language-agnostic ASTs to find function declarations including their signature. This information is sent asynchronously to our LSP client, which uses it for an example wizard that can be triggered through a code lens. Code lenses are part of the LSP, supported by VS Code, and allow the annotation of source with links to trigger certain actions. In our case, we annotate function declarations with an "Add Example" action that opens appropriate UI components in VS Code to define new examples. This way, it is possible to set a name, the probe mode, and all parameters and input values. After the user has defined a new example, the example definition is sent to the extended TruffleLSP server, which generates the example definition, modifies the file accordingly, runs the examples, and informs the client about the modification. To the user, this operation is transparent. It is, however, necessary to perform the modification through the server, because it has additional AST information required for correctly placing examples.

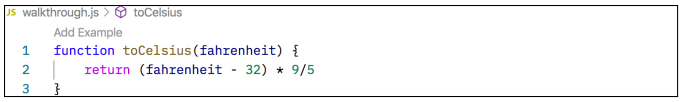
6 Walkthrough and Polyglot Scenario

In the following, we walk the reader through our language-agnostic Babylonian Programming system and give an example for how it can be used for polyglot programming.

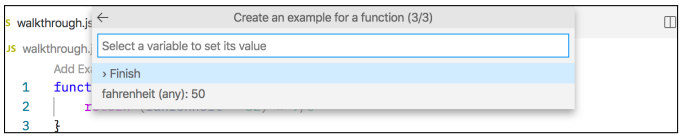
6.1 Walkthrough

In this section, we walk the reader through an example workflow using the series of screenshots from Figure 3. The example is inspired by the code in Listing 1.

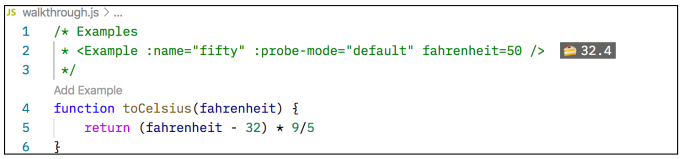
Figure 3a shows the starting point: A function for converting Fahrenheit to Celsius that the user has developed. The function is annotated with our "Add Example" code lens. Instead of a unit test, the user decides to use an example by clicking on the code lens. The example wizard is opened and the user is prompted to provide a name, probe mode, and an example value for the `fahrenheit` parameter (Figure 3b). After accepting the new example, a corresponding code comment is generated and added to the `toCelsius()` function. Almost instantly, the example in line 2 is annotated with the calculated result (Figure 3c). 50 degrees Fahrenheit, however, are 10 degrees Celsius, not 32.4. The user therefore decides to add an appropriate assertion on the return statement (line 5 of Figure 3d). As expected, the assertion fails for the given example, which is visualized with an X mark. The user now adds a probe with `fahrenheit - 32` as expression (line 5 of Figure 3e). The system almost immediately reports that the probe result is 18, which is expected. Therefore, the multiplier in line 7 must be incorrect. The user realizes the multiplier needs to be inverted, after which the assertions succeeds as visualized with a check mark. Also, the correct result is displayed in the decoration of the example in line 2 of Figure 3e. Finally, the user may decide to use `toCelsius()` in another function of the program, such as the `getTemperatureText()` function (line 2 of Figure 3f). An example can also be added to this function, which not only reveals what it returns for `fahrenheit=80` in the corresponding example decoration. The `probe` in line 10 is also active for this additional example, as one can see in the annotation in line 10: It shows that the value is 48 for the example of the `getTemperatureText()` function and 18 for the example



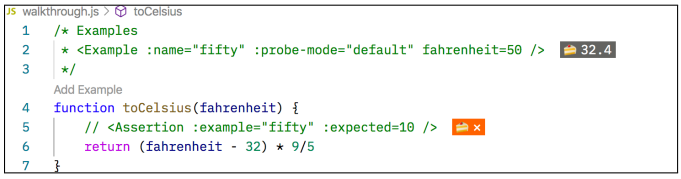
(a) Starting point: a Fahrenheit-to-Celsius conversion function written in JavaScript.



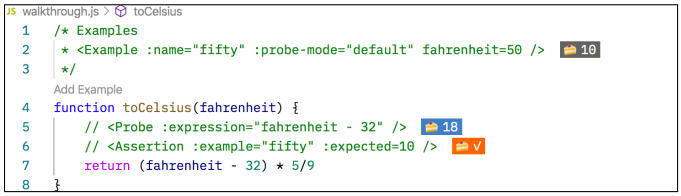
(b) Defining an example using the wizard.



(c) Code with an example definition revealing a problem in the Fahrenheit-to-Celsius conversion.



(d) Sample code with an additional assertion.



(e) Refactored and fixed code with an additional probe ensuring the conversion works correctly for the given example.

(f) Extended code with a `getTemperatureText()` function and a second example revealing that the Celsius value is not rounded.

Figure 3. Screenshot series of VS Code running our GraalVM extension with Babylonian Programming capabilities. The series illustrates the evolution of the code as well as the feedback by the system.

declared on `toCelsius()`. After seeing the result of the example with the name "warm", the user may want to round the Celsius value in either function to ensure that the string returned by `getTemperatureText()` is nice and short.

6.2 Example-Based Live, Polyglot Programming

Implementing a Babylonian Programming system in a language-agnostic way not only avoids the costs of having to implement the same mechanisms from scratch for each language ecosystem. It also makes it possible to leverage the system for polyglot programming where an application can be built using multiple languages. The main premise of polyglot programming is to foster software reuse by enabling programmers to use the best language, library, or framework for the job.

Figure 4 shows a polyglot example program developed in our Babylonian Programming system using JavaScript, Ruby, and a third language called SimpleLanguage, which is Truffle's reference language implementation. The JavaScript file contains a `getTemperatureText()` function similar to the example in Figure 3. This time, however, the function takes a city as argument and returns a formatted string with the city and corresponding temperature information as result. The function is annotated with two examples in line 2 and 3, "London" and "San Francisco". In the first line of the function, a probe on the `city` variable is set and displays the expected city names for each example. In line 7, GraalVM's polyglot API is used to call out to Ruby, which downloads a JSON-formatted file synchronously from openweathermap.org for the given city. In the same line, the result is passed into JavaScript's `JSON.parse()` function to turn the JSON content into a JavaScript object. To ensure that the temperature information for London in the UK, not Ontario is found, an assertion is present in line 8. Note that this assertion is example-specific and therefore not checked for the "San Francisco" example. In line 9, the JavaScript code calls out to Ruby again, this time evaluating the `render.rb` file displayed in the middle of the screenshot. Since the file returns a lambda-like Ruby `Proc`, it can be called with `city` and `fahrenheit` parameters. The result for each example is displayed in their decorations in line 2 and 3 of the `JS` file. The Ruby file, in turn, converts the `fahrenheit` parameter using the `toCelsius()` routine written in SimpleLanguage. This routine also contains a probe that is active for both examples. This probe reveals that SimpleLanguage is so simple that it does not support floating-point numbers. Therefore, the program continues with imprecise values from this point on. For comparison, we have added a probe with an expression to the `render.rb` file in line 5 that calculates the precise Celsius values in Ruby. Additionally, the following line contains another probe, this time with an erroneous expression. Instead of letting the instrumentation fail entirely, our Babylonian Programming system is able to display an appropriate error message as decoration instead. In line 7 of the `render.rb` file,

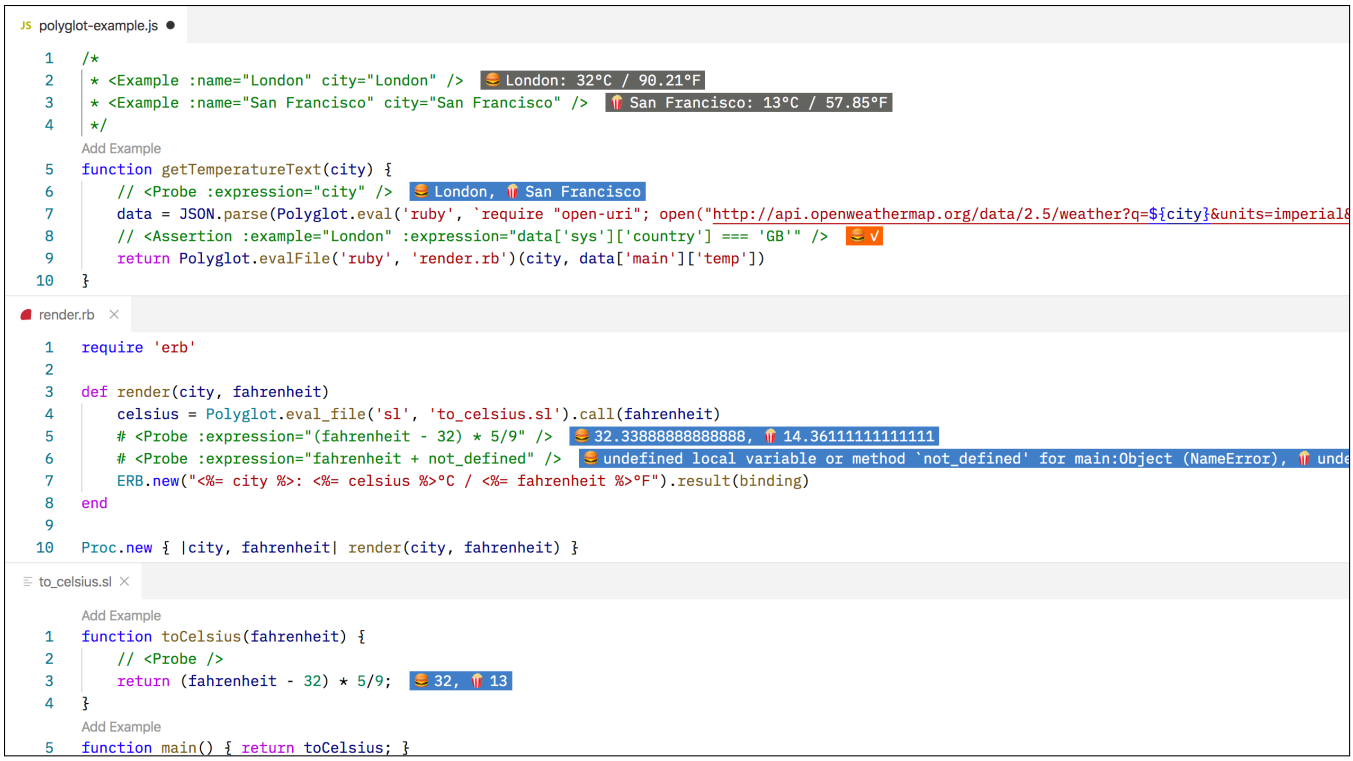


Figure 4. Exemplified polyglot code to determine temperature information for a given city using JavaScript (JS), Ruby, and SimpleLanguage (SL). The probes set in Ruby and SL are triggered by the examples defined in JS.

the program uses Ruby's templating language to render the string that is finally returned by `getTemperatureText()`.

This constructed polyglot example allows us to make a number of observations about our Babylonian Programming system and GraalVM's approach to polyglot programming: First, it demonstrates that the system can indeed be used throughout different programming languages at the same time. We also saw that the instrumentation is robust and capable of dealing with errors. In addition, the `evalFile` function of the polyglot API requires language-specific tricks, such as the Ruby `Proc`, to return something useful. Instead, it might be more convenient to load and import foreign code through the module system of a language. Moreover, we were able to observe misbehavior in the program introduced by the use of SimpleLanguage. One could argue that it is not intended to be used to build sophisticated software. Nonetheless, we believe this misbehavior is a representative example for the kind of issues that can easily be encountered during polyglot programming. Live tools such as our Babylonian Programming system can help programmers to better understand different language semantics and consequently produce less errors when mixing software from different language ecosystems. Lastly, we discuss the limitations of our system with regard to polyglot programming in detail in Section 7.3.

7 Discussion

The main features of ELP is the feedback loop resulting from making any function in a system executable through annotated examples, and providing fine-grained feedback directly within the IDE or code editor. Our prototype demonstrates that it is possible to provide this feedback loop using a language-agnostic infrastructure. The major obstacles to implementing a richer UI beyond the DSL result from the limited set of UI concepts of the target programming environment. The instrumentation level can provide all dynamic information for current ELP features.

In this section, we discuss our approach and implementation with regard to the provided ELP features and observed limitations. We further describe whether the limitations pose fundamental issues to a language-agnostic implementation of ELP, whether they are a result of our implementation, or whether they are merely left out of the prototypical implementation.

7.1 Comparison of Features

Table 2 shows to which extent our prototype implements ELP features by comparing the implemented features of the prototype with the features of the original Babylonian Programming system and the example-centric programming

Table 2. An overview of the features of the example-centric environment [1], the original Babylonian Programming system, and our language-agnostic prototype. Additionally, the overview lists the respective obstacles to implementing missing features. The list of features is based on an overview of features of ELP systems [23]. We distinguish between obstacles on the side of the user interface and instrumentation obstacles. Obstacles described as "conceptually challenging" are difficult to reconcile with a language-agnostic implementation in general, "technically challenging" means that the feature is difficult to implement with the chosen technologies, "not yet implemented" means that the features is simply not implemented yet, "✓" means that no further work is required.

	support for feature			obstacles	
feature	example-centric env.	original implementation	our prototype	user interface	instrumentation
granularity of feedback	statement-level	probe on any expression in program code	statement-level, probes with own expression	✓	✓
state over time	navigate to points in time	states of probe inline, explicit state transitions, sliders to navigate	states of probe inline	sliders: technically challenging	state transitions: technically challenging
state over modules	navigating the full trace	all probes throughout the system show results	all probes throughout the system show results	✓	✓
arbitrary objects	not explicitly covered	object identity through emoticons, complex state in probe widget, object inspector	complex state as "display string"	object identity: not yet implemented, object inspector: not yet implemented	object inspector: not yet implemented
domain-specific feedback	none	none	none	conceptually challenging	✓
multiple examples	multiple implicit examples, selection through trace	multiple explicit examples, named examples, individually enabled	multiple explicit examples, named examples, individually enabled	embedded graphical editors: technically challenging	✓
reusing example parts	through variables in the program	through named instance templates, links to objects	none	not yet implemented	✓
behavioral highlighting	executed statements per example (trace)	executed statements per example (trace)	executed statements per example (trace)	✓	✓
specifying context	explicit setup code before implicit example	automatically executed pre- and postscripts, replacements, executions partially isolated	executions are isolated in sandbox	pre- and postscripts, replacements: not yet implemented	pre- and postscripts, replacements: not yet implemented
keeping track of assumptions	locations of exceptions, assertions in trace view	shows erroneous examples	shows erroneous examples, example-specific assertions inline	✓	✓
navigating the trace	full trace view for navigation	sliders for iterations and activations	none	sliders: technically challenging, trace overview: not yet implemented	✓

environment, which is a prominent and early ELP environment [1].

The example-centric environment is not based on probes but rather on a fully recorded trace (as an optimization, only these parts of a system are traced which are currently visible to users). As a consequence, while providing a similar experience, the available features are different from implementations of the Babylonian Programming system.

In general, our prototype can provide the fundamental live feedback loop of ELP systems. Programmers can specify and activate multiple examples through the DSL and they can get feedback on all results of selected statements throughout the entire system. Complex objects are currently supported through the `toDisplayString` message provided by Truffle's `InteropLibrary`, which retrieves a human readable string representation for any object. Similar to the example-centric system, users can also add example-specific assertions to quickly discover violated assumptions. Beyond feedback on the results, the system can visualize which lines were actually executed by showing probes for all statements, and it can display which examples caused an exception.

Three major features are currently missing from our prototype: navigating the trace, inspecting complex objects, and specifying context beyond an isolated execution environment. The specification of context and an inspector for complex objects are possible but not implemented yet due to time constraints. The trace navigation as supported by the original Babylonian Programming system faces more fundamental challenges resulting from the features of VS Code. We discuss the way forward and the underlying obstacles in the following paragraphs.

7.2 Missing UI Concepts

A major obstacle to a richer user interface are the UI concepts provided by the programming environment. Two major user interface concepts that were available in the original Babylonian Programming system are missing from our prototype due to this (see Table 2): sliders and embedded graphical editors.

The first limitation is missing UI sliders, which other Babylonian Programming systems, such as Babylonian/JS, provide to allow users to scroll through loops and recursive function calls. We were unable to add sliders through the VS Code extension as VS Code only allows a predefined set of UI components in the editor, none matching the functionality of a slider. Instead, we could have introduced another kind of annotation in the DSL or an overview of the complete trace in an additional panel.

The second limitation results from the same underlying reason. Due to the restricted set of UI components in the VS Code editor, our system is unable to provide embedded graphical editors for instance templates, pre- and postscripts, assertion annotations, probes with their own expression, or replacements. In general, programmers could modify all of these through the DSL, thus making the features available. At the same time, graphical tool support might lessen the cognitive overhead of switching between the programming language and the DSL and is therefore desirable.

In summary, the main obstacle for a more expressive user interface, at the time of writing, are the visualization capabilities available to current LSP clients. VS Code seems to provide the best coverage of the features of the LSP. For example, VS Code appears to be the only LSP client with proper support for code lenses. The reason for this might be that the LSP and VS Code are both maintained by Microsoft, which is also the reason why we chose to implement our approach based on them. However, the amount of UI components we could use for our implementation was still quite limited: Although VS Code's UI is makes use of HTML and JavaScript, extensions do not seem to be allowed to add custom HTML-based components. Instead, we were limited to `setDecorations` and code lenses.

All these limitations, however, are technical issues imposed by our decision to use VS Code as client. Other multi-purpose IDEs or code editors with LSP clients[3], such as Eclipse, Atom, or Sublime Text, might not come with such restrictions and might, for instance, provide graphical editors for examples or annotations. These modifications would, however, again be environment-specific.

Nevertheless, these UI limitations do not constrain the key contribution of ELP which is the live feedback loop that results from adding concrete examples to code and being able to see intermediate runtime states directly within the source code. By providing ELP features through the textual DSL, we might get a less convenient UI, but we gain support for the ELP experience in a variety of programming environments.

Other UI Limitations. A fundamental limitation is the lack of environment-agnostic mechanisms to provide "domain-specific feedback". Through providing custom visualizations for domain-objects, probes might become more useful, for example by displaying objects representing angles as a direction vector. This feature is also missing from most other ELP systems [23]. However, supporting this feature as part of the environment-agnostic part of the ELP implementation might not be possible. Any mechanism to provide domain-specific visualizations of runtime values will have to be provided by the implementers of the extension of the target environment and cannot be shared between environments.

In addition, the code decorations we use display the result of each example in a single line. This is a minor limitation. While it is possible to horizontally scroll through all of them in VS Code, it could be hard to find the right result in case many examples are active at the same time. Users, however, are in full control over the number of active examples.

[3] See https://git.io/JJHaO for a list of tools supporting the LSP (accessed 2020-08-12)

7.3 Instrumentation and GraalVM

Apart from UI limitations on the side of the client, our implementation has some limitations with regard to program instrumentation and GraalVM.

Replacements as well as pre- and postscripts have not been implemented yet due to time constraints. Both pose no fundamental challenges. For example, replacements, could be implemented in Truffle's instrumentation framework by throwing an unwind throwable (e.g. `throw context.createUnwind(alternativeReturnValue)`) in the `onEnter` hook of our `ExecutionEventNodeFactory`. This would not only force a different return value, it would also avoid the execution of the replaced node entirely, which is the expected behavior for replacements. Again, a graphical tool would be the more challenging part for supporting replacements within an IDE.

A more challenging limitation are probes attached to variable assignments as supported by previous implementations of the Babylonian Programming system. These probes can show the value of a variable before and after the assignment. Currently, probes are attached to nodes denoted with the `StatementTag`, which does not provide access to a potential variable assignment. In Python, for example, assignments always return `None`. Through using the `WriteVariableTag` it would be possible to determine the value of the variable before and after the assignment. However, the `WriteVariableTag` is a new part of the instrumentation framework therefore not fully supported by GraalVM languages.

The runtime must be able to provide timely feedback. Therefore, instrumentation must not decrease the run-time performance of a program significantly. Since run-time performance is a primary goal of GraalVM and all of its APIs, we believe it is a good fit for building ELP systems. A preliminary response time analysis of our system can be found in Appendix A.

In addition, our system is currently limited to JavaScript, Ruby, and SimpleLanguage, because we rely on a relatively new API and we are one of the first tool builders to use it. At the time of writing, only JavaScript is officially supported by GraalVM. SimpleLanguage is a reference toy language not intended to be used in production. And all other languages, including Ruby, are considered experimental.

Moreover, we identified numerous inconsistencies in the support for both instrumentation and interoperability of GraalVM languages. Figure 4, for example, shows that Ruby does not currently support to find function declarations yet. This is the reason why no code lenses are displayed in the Ruby part of the program. Moreover, the `StatementTags` are not fully or correctly supported by all GraalVM languages yet, avoiding some probes and assertion from triggering. In terms of interoperability issues, we found that the implementation of certain `InteropLibrary` messages, such as `toDisplayString()`, are missing or inconsistent across languages.

All of these limitations and issues are not fundamental. The GraalVM team is aware of the mentioned shortcomings and we are working with them to resolve them in a future version of GraalVM. When this is the case, our Babylonian Programming system will work consistently across all GraalVM languages.

Architectural Considerations. Our approach demonstrates that the run-time information required by a Babylonian Programming system can be fully provided in a language-agnostic way by the Truffle instrumentation framework. We believe that an implementation using Truffle is superior to the code rewriting approach used in Babylonian/JS and found that is just as elegant as the approach based on Context-Oriented Programming (COP) in Babylonian/S, but potentially capable of providing better performance. In the latter case, the support for COP is also quite language-specific, which we were able to avoid by implementing the mechanism in a language implementation framework rather than in a specific language implementation.

Furthermore, the LSP can be easily extended with hooks needed to implement a Babylonian Programming system without breaking compatibility with other LSP clients. Our implementation shows that this is relatively easy to do, which may encourage other tool builders to base their future tooling on Truffle and the LSP as well.

7.4 ELP and Polyglot Programming

Being able to use one tool across different languages avoids the costs of learning language-specific tools for programmers and improves the programming experience by making it more consistent. In the context of polyglot programming, a Babylonian Programming system can help programmers better understand the effects imposed by mixing different languages, such as differences in semantics. In the simple polyglot example shown in Figure 4, the system instantly revealed that SimpleLanguage does not support floating-point numbers and that such numbers are rounded down to natural numbers when passed to SimpleLanguage. We believe that our Babylonian Programming system can therefore help to make polyglot programming more approachable to programmers. Further evaluation in the form of a user study, however, is still needed.

7.5 The Future of Rich, Language-Agnostic Tooling

The LSP is designed to decouple language-specific tool implementations from the corresponding graphical UI in IDEs to provide more flexibility to programmers. This is done by providing information about the code in question for code completion, goto definitions, and other common IDE features. This information is often obtained though static code analysis by an LSP server. As the TruffleLSP project

has demonstrated, features provided through the LSP can be further enhanced with dynamic run-time information (e.g. code completion) and new ones can be added (e.g. code coverage) [28].

In the future, we would like to see this idea being pushed even further by supporting more programming features that rely on dynamic run-time information. Some of these features, for example an infrastructure for building object inspection tools, are already provided by other protocols, such as the Chrome DevTools Protocol [5]. That similar protocols can be implemented in a language-agnostic way has also been demonstrated by GraalVM, which comes with built-in support for the Chrome DevTools Protocol since its first stable release. Combining the LSP with such a debugging protocol would therefore enable further IDE features that help programmers to understand the run-time behavior of their programs.

Nonetheless, our community has explored various other approaches to improve the programming experience. As this work has demonstrated, it is possible to integrate a live feedback loop into the LSP. Making such a loop officially part of the protocol would encourage all LSP clients to add support for live programming. In particular, we believe the LSP would benefit from the ability to evaluate code interactively and to instrument programs. In some ways, this is what Jupyter kernels [11] enable in notebooks: Provide a backend for interactive code execution.

Even if full, first-class ELP support may not be added to LSP, supporting a richer set of UI concepts might make powerful tools more reusable and as a consequence more feasible. For example, as our prototype hints, adding LSP requests for generic decorations with full HTML support would already enable many of the features of live probes.

Ultimately, if we manage to extend the LSP in any of these ways, we believe it would be possible to enable more live and exploratory programming features in today's commonly used programming environments.

8 Related Work

The goal to provide reusable tooling across programming languages or programming environments is part of many approaches.

Various execution environments expose information through APIs which that can be used to create debugging tools, for example the Java Debug Interface (JDI) [22] or the Python debugger framework library `bdb` [3]. Beyond these APIs, there are also protocols defined between programming and execution environments, similar to the LSP. These protocols aim to decouple the implementation of tools from the user interface even further. A contemporary example of such a protocol is Microsoft's Debug Adapter Protocol (DAP) [17]. The DAP aims to provide a common protocol between user interfaces for debuggers and language-specific debugger implementations, thereby making both sides easier to reuse. Besides full protocols for specific tools, there have also been efforts towards reusable data formats, for example the OPEN.xtrace format for execution traces [21].

A different approach that can also provide language-agnostic tooling are language workbenches [4]. Language workbenches aim to support programmers with developing new languages and the corresponding development environments. Based on the language definition, many workbenches can provide static tool support, such as syntax highlighting, code navigation, or refactoring [2]. While the resulting tool implementations are bound to a specific language, the code for generating these tools is language-agnostic.

Finally, there are a number of approaches directly built upon the Truffle infrastructure. The initial work that presented Truffle's instrumentation framework demonstrated how this API can be used to build fast, language-agnostic tools such as debuggers, code coverage tools, and tools for dynamic program analysis [30]. Based on this infrastructure GraalVM, for example, provides support for the Chrome DevTools Debugger Protocol across all supported languages, a code coverage tool, and various profiling tools. None of the tools, however, aim at providing live feedback in the sense of ELP or exploratory programming. Moreover, they assume good performance based on previous work on self-optimizing AST interpreters [34]. A later evaluation study with a prototypical Ruby debugger showed that debugging tools on top of the instrumentation framework are fast [27], another study showed the same for dynamic program analysis for Node.js [29]. Our work demonstrates that this infrastructure can also be used to build tools for live feedback and therefore enable ELP.

9 Conclusion and Future Work

We presented an approach for building an ELP system, more specifically the Babylonian Programming system, in a language-agnostic way based on GraalVM's Truffle framework and the LSP. We demonstrated that both Truffle and the LSP meet the requirements for enabling a language-agnostic implementation of an ELP system. Such an implementation keeps the cost of tool building low without sacrificing its core functionality. Furthermore, it allows programmers to use the tools in a more consistent way across languages, and in the context of polyglot programming.

We have implemented our approach based on GraalVM's TruffleLSP and its extension for VS Code. We have demonstrated how this implementation is able to bring ELP to VS Code and explained how the LSP specification could be extended to make our approach fully environment-agnostic. We gave examples for how our system can be used and how it can help to build polyglot applications. And we discussed advantages and limitations of our approach, based on what we have learned from our prototype.

Apart from some features that we could not implemented due to time constraints, we identified several problems in GraalVM languages that must be addresses to make the instrumentation framework and hence our tooling work correctly. Additional directions for future work include exploring other LSP clients and corresponding IDEs and code editors with regard to appropriate UI components for visualizing dynamic run-time information. Based on the results, an official proposal for extending the LSP specification could be submitted. Future work may also investigate if our language-agnostic tooling can be used for static languages such as C or C++ or for languages that are not file-based such as Squeak/Smalltalk. Future work could also aim at finding out to what extent our approach can be applied to other language implementation frameworks such as RPython.

We hope this work encourages other tool builders to think about language-agnostic approaches for their tools, which can keep development costs low and help to make them available to a broader audience.

Acknowledgments

We gratefully acknowledge the financial support of HPI's Research School[4] and the Hasso Plattner Design Thinking Research Program[5].

A Response Time Analysis

Table 3. Number of source lines of code (SLOC) per language, examples, probes, and assertions of the three benchmarks.

	#1 Example	#2 Toy	#3 Polyglot Toy
SLOC	7 JS	207 JS	207 JS + 11 SL
Examples	2	10	10
Probes	2	100	101
Assertions	1	100	100

To evaluate the usability of our implementation further, we analyzed the response time of the initial version of the system based on GraalVM 20.0.0 using three micro-benchmarks. In practice, we expect the response time to be dominated by the duration of the execution of the example which very much depends on the application. Thus, in these benchmarks, we focus on getting a first impression of the magnitude of the overhead imposed by our tooling infrastructure. The benchmark results were taken from our prototype system, which focuses on functionality and does not aim at performance yet, and should therefore be taken with a grain of salt.

For the response time analysis, we measured the adaptation and emergence phases [24] for the example program from Section 6.1 (#1) as well as for two additional toy programs. Both toy programs are based on a JavaScript file that

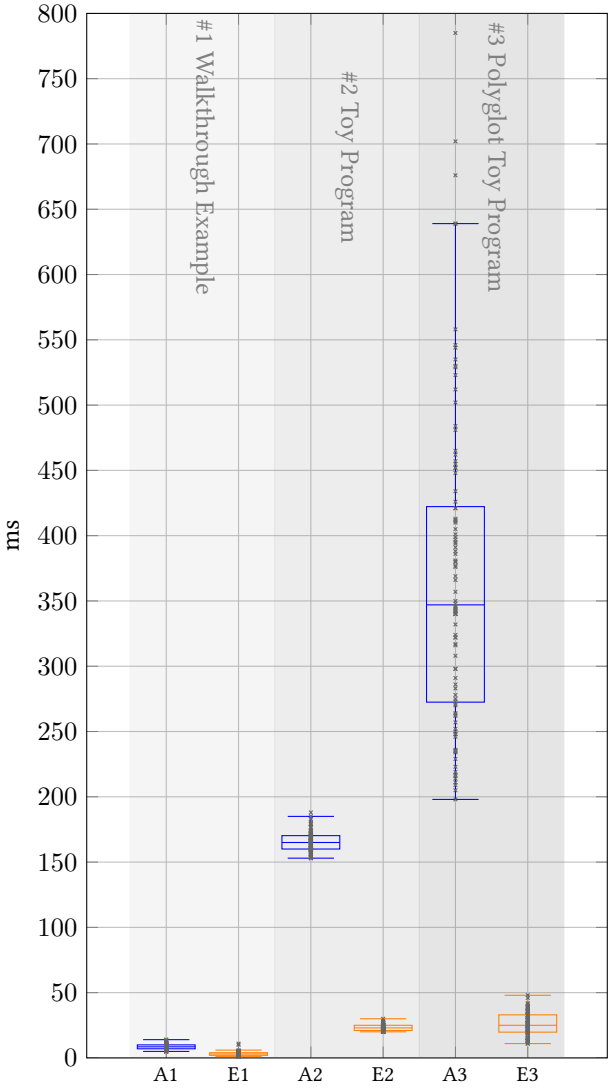


Figure 5. Adaptation (A1, A2, A3) and Emergence (E1, E2, E3) analysis of three different benchmark programs, 100 data points each. The ends of the whiskers represent the lowest and highest datum within 1.5 IQR of the corresponding quartile.

applies an operator inline 100 times to a given number with probes and another 100 times with assertions. In the first toy program (#2), the operator increments the number randomly. In the other one (#3), we use an operator written in SimpleLanguage, which turns the toy program into a polyglot one crossing the language boundary 200 times for the operator. This SimpleLanguage operator is based on the `toCelsius()` function from Figure 4, includes a probe, and returns zero if the result exceeds 4000. Table 3 lists the characteristics of each program. For the benchmark, we define the adaptation phase as the time between a `didChange` or `didOpen` event is triggered by the user in the code editor until the server has

[4] https://hpi.de/en/research/research-school.html
[5] https://hpi.de/en/dtrp/

compiled the modified code. The latter moment is also when the emergence phase starts: right before the code is executed in the context of our Truffle instrument. It ends right after the `setDecorations` API call was triggered in the editor. In order to reduce the influence of external factors, we produced 100 data points for each benchmark by changing a constant value randomly by hand. All benchmarks were performed on a 13-inch MacBookPro15,2 (CPU: 2.7 GHz Quad-Core Intel Core i7; Memory: 16 GB 2133 MHz LPDDR3). We used a custom-built GraalVM Community Edition without libgraal, which would offer faster JIT-compilation, and with VS Code in extension development mode. To create a realistic scenario and get an initial impression of response times to be expected during programming, we did not close any other applications, such as IDEs and terminals, that were open on our development machine during each run.

Figure 5 shows the benchmarks results. The times of the adaptation phase increase much more rapidly with the complexity of the programs than the times of the emergence phase. Nonetheless, the system is able to provide feedback in under 20ms (mean(A1) + mean(E1)) for the first, under 200ms for the second, and under 400ms for the third program. All three benchmark programs run under one second, which is considered the threshold after which a user starts wondering if an interactive system is still responding [8, p. 163]. In case of errors and timeouts, our system is still able to respond. Therefore, we conclude that it may be feasible to apply our approach in the context of Truffle and GraalVM.

References

[1] Jonathan Edwards. 2004. Example centric programming. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 124. https://doi.org/10.1145/1028664.1028713

[2] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings (Lecture Notes in Computer Science)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.), Vol. 8225. Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[3] Python Software Foundation. 2020. bdb — Debugger framework. https://docs.python.org/3/library/bdb.html Accessed: 2020-03-20.

[4] Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? https://martinfowler.com/articles/languageWorkbench.html Accessed: 2020-05-20.

[5] Google. 2020. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/ Accessed: 2020-03-20.

[6] Chris Granger. 2014. *Light Table.* http://lighttable.com Accessed: 2020-05-20.

[7] Tomoki Imai, Hidehiko Masuhara, and Tomoyuki Aotani. 2015. Shiranui: a live programming with support for unit testing. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 36–37. https://doi.org/10.1145/2814189.2817268

[8] Jeff Johnson. 2014. *Designing with the Mind in Mind, Second Edition: Simple Guide to Understanding User Interface Design Guidelines* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[9] JSON-RPC Working Group. 2020. *JSON-RPC 2.0 Specification.* https://www.jsonrpc.org/specification Accessed: 2020-05-12.

[10] Saketh Kasibatla and Alessandro Warth. 2017. *Seymour: Live Programming for the Classroom.* https://harc.github.io/seymour-live2017/ Accessed: 2020-05-22.

[11] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. 2016. Jupyter Notebooks ? a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, Fernando Loizides and Birgit Scmidt (Eds.). IOS Press, 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

[12] Donald E. Knuth. 1972. Ancient Babylonian Algorithms. *Commun. ACM* 15, 7 (1972), 671–677. https://doi.org/10.1145/361454.361514

[13] Andrew J. Ko and Brad A. Myers. 2009. Finding Causes of Program Output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, USA) *(CHI '09)*. Association for Computing Machinery, New York, NY, USA, 1569–1578. https://doi.org/10.1145/1518701.1518942

[14] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan O. Borchers. 2014. How live coding affects developers' coding behavior. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*, Scott D. Fleming, Andrew Fish, and Christopher Scaffidi (Eds.). IEEE Computer Society, 5–8. https://doi.org/10.1109/VLHCC.2014.6883013

[15] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik (Eds.). ACM, 1–7. https://doi.org/10.1145/3313831.3376494

[16] Sean McDirmid. 2013. Usable live programming. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (Eds.). ACM, 53–62. https://doi.org/10.1145/2509578.2509585

[17] Microsoft. 2020. Debug Adapter Protocol. https://microsoft.github.io/debug-adapter-protocol/ Accessed: 2020-05-20.

[18] Microsoft. 2020. Language Server Protocol. https://microsoft.github.io/language-server-protocol/ Accessed: 2020-03-02.

[19] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. GraalSqueak: Toward a Smalltalk-Based Tooling Platform for Polyglot Programming. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 14–26. https://doi.org/10.1145/3357390.3361024

[20] Fabio Niephaus, Tim Felgentreff, Tobias Pape, Robert Hirschfeld, and Marcel Taeumel. 2018. Live Multi-language Development and Runtime Environments. *The Art, Science, and Engineering of Programming* 2, Article 8 (2018), 30 pages. Issue 3. https://doi.org/10.22152/programming-journal.org/2018/2/8

[21] Dusan Okanovic, André van Hoorn, Christoph Heger, Alexander Wert, and Stefan Siegl. 2016. Towards Performance Tooling Interoperability: An Open Format for Representing Execution Traces. In *Computer Performance Engineering - 13th European Workshop, EPEW 2016, Chios, Greece, October 5-7, 2016, Proceedings (Lecture Notes in Computer Science)*, Dieter Fiems, Marco Paolieri, and Agapios N. Platis (Eds.), Vol. 9951. Springer, 94–108. https://doi.org/10.1007/978-3-319-46433-6_7

[22] Oracle. 2020. Java Debug Interface. https://docs.oracle.com/javase/10/docs/api/jdk.jdi-summary.html Accessed: 2020-03-20.

[23] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming - Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *Programming Journal* 3, 3 (2019), 9. https://doi.org/10.22152/programming-journal.org/2019/3/9

[24] Patrick Rein, Stefan Lehmann, Toni Mattis, and Robert Hirschfeld. 2016. How Live Are Live Programming Systems? Benchmarking the Response Times of Live Programming Environments. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop* (Rome, Italy) *(PX/16)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/2984380.2984381

[25] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2019. Implementing Babylonian/S by Putting Examples Into Contexts: Tracing Instrumentation for Example-Based Live Programming as a Use Case for Context-Oriented Programming. In *Proceedings of the Workshop on Context-Oriented Programming* (London, United Kingdom) *(COP '19)*. Association for Computing Machinery, New York, NY, USA, 17–23. https://doi.org/10.1145/3340671.3343358

[26] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *Programming Journal* 3, 1 (2018), 33. https://doi.org/10.22152/programming-journal.org/2019/3/1

[27] Chris Seaton, Michael L. Van De Vanter, and Michael Haupt. 2014. Debugging at Full Speed. In *Proceedings of the Workshop on Dynamic Languages and Applications* (Edinburgh, United Kingdom) *(Dyla'14)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2617548.2617550

[28] Daniel Stolpe, Tim Felgentreff, Christian Humer, Fabio Niephaus, and Robert Hirschfeld. 2019. Language-independent development environment support for dynamic runtimes. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2019, Athens, Greece, October 20, 2019*. 80–90. https://doi.org/10.1145/3359619.3359746

[29] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient Dynamic Analysis for Node.Js. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/3178372.3179527

[30] Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *Programming Journal* 2, 3 (2018), 14. https://doi.org/10.22152/programming-journal.org/2018/2/14

[31] Tijs van der Storm and Felienne Hermans. 2016. Live Literals. Presented at the Workshop on Live Programming (LIVE) 2016. https://homepages.cwi.nl/~storm/livelit/livelit.html Accessed: 2020-05-20.

[32] Bret Victor. 2012. Inventing on Principle. Presented at the the Canadian University Software Engineering Conference (CUSEC) 2012. https://vimeo.com/36579366 Accessed: 2020-05-20.

[33] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. 187–204. https://doi.org/10.1145/2509578.2509581

[34] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587