Gramada

Immediacy in Programming Language Development

Patrick Rein[⋆, †] Robert Hirschfeld[⋆, †] Marcel Taeumel[⋆, †]

[⋆] Hasso Plattner Institute, University of Potsdam

[†] Communications Design Group (CDG), SAP Labs, USA; Viewpoints Research Institute, USA

first.last@hpi.uni-potsdam.de

Abstract

Domain-specific languages (DSLs) improve software maintainability and programmer productivity by making domain concepts more explicit in the code. However, developing syntax and semantics of DSLs is often challenging and as a result developers seldom take advantage of the benefits of DSLs. One way to lower the entry barrier to DSL development is to give developers immediate and continuous feedback on modifications to a language. We propose Gramada, an environment for developing DSLs in Squeak/Smalltalk which is designed to provide such a live programming experience. It is based on a language development framework with additional support for incremental compilation to improve system response times and a set of tools which creates a steady frame and allows programmers to quickly explore changes to the syntax of a language. Our benchmarks and discussion illustrate how Gramada can give visual feedback on most changes to the language in a way that supports live programming. We conclude that Gramada's feedback is fast and consistent enough to make exploring the effects of changes a lively and productive activity during the interactive development of DSLs.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]

Keywords domain-specific languages, language development, immediate feedback, tools, incremental compilation

1. Introduction

Domain-specific languages (DSLs) are a software engineering technique in which programmers develop and use a dedicated language for a particular area of interest. Often, the goal of these languages is to increase the productivity of programmers or to allow domain experts to express, understand, or modify domain-specific code [7, 20]. To fit syntax and semantics as close to the domain as possible, programmers might consider implementing a DSL as a new language, instead of, for example, implementing it as a domain-specific interface to a library. While this allows for more freedom regarding language design, it also requires a full language implementation, traditionally consisting of tools like parsers, compilers and interpreters. To ease this development activity and thereby making such DSLs more feasible, a number of frameworks and systems were created. These range from mere libraries providing parsing algorithms to complete environments supporting *language-oriented programming*, which makes DSL development a primary development activity next to application development.

Nevertheless, outside of the programming language community, language development is often considered difficult and hence application developers seem to seldom develop and use appropriate DSLs. In order to make DSL development more approachable, it should be easy to explore and understand DSL implementations. One general way to improve understanding of programmers is providing them with immediate and continuous feedback on the effects of their changes to the system [10, 17]. This might help closing the cognitive gap between the static source code and the dynamic behavior of a system [21].

One class of development systems, nowadays referred to as *live programming systems*, promotes programming with continuous and immediate feedback on changes to the source code. Such systems, as Lisp [18], Smalltalk [8], or Self [32], allow the developer to constantly see the effects of changes to the source code in the behavior of the system running side-by-side with the development environment. Since these live programming environments are designed for general-purpose programming, they could also support the development of DSLs. However, they mostly focus on general application development, and thus do not provide dedicated support for developing languages.

Onward!'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4076-2/16/11...$15.00
http://dx.doi.org/10.1145/2986012.2986022

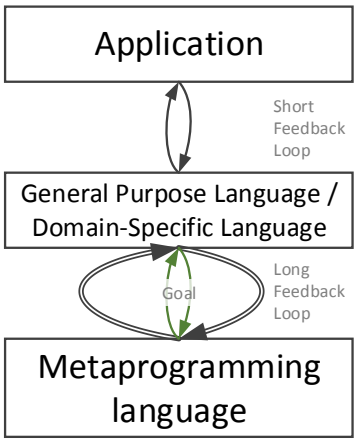

Figure 1. Live programming systems provide immediate and continuous feedback for application development. However, specialized language development environments for developing languages are often not designed to provide such feedback. Thus, we propose Gramada, a development environment which provides immediate feedback through short response times and continuous feedback through appropriate tools.

In contrast, specialized language development environments currently often focus on producing efficient language implementations and do not focus on providing a live programming experience (see Figure 1). Often, they require an explicitly initiated translation step from a language definition to an executable form which often creates a completely new executable. Thus, it delays feedback on the impact of the modifications made to the language, as the developer can only execute and evaluate the change afterwards. Further, developers have to manually reload the new executable or re-execute sample programs, to see any effects. This re-execution might discard any previous execution or exploration and thus can break the continuity of the mental model of the developer. Preserving the context through continuous feedback and shortening the delays of feedback might bring the benefits of live programming to the domain of language development.

In this paper, we show how to take advantage of live programming in the particular activity of specifying the syntax of DSLs. We demonstrate this through tools and mechanisms implemented in the proposed language development environment *Gramada*. The implemented features show how the engineering techniques enabling immediacy in application-level development also work for language development. In particular, our contributions are:

- The language development library Ohm/S which is a Smalltalk adaptation of the library Ohm [4, 35]. Ohm/S is adjusted to enable immediate and continuous feedback through incremental compilation of syntax definitions and dynamic dispatch of grammar rules.
- The Gramada tool set which allows DSL developers to see effects of changes to the syntax definition of a language quickly and without explicitly triggered re-executions. It includes editors, an input sandbox, a visualization of parse results, an interactive debugger, and an SUnit extension for syntax tests.
- An implementation of Ohm/S and the tool set in Squeak/Smalltalk [11, 27] based on the data-driven tool development environment Vivide [28, 29].
- A quantitative evaluation of the feedback loops regarding the durations of modifications of a language and the time Gramada tools need to display changed behavior. We also qualitatively discuss how Gramada preserves the context through continuous feedback.

In the following, we first discuss in Section 2 the benefits of live programming and the resulting requirements for a programming environment. Based on this background, Section 3 describes the single tools in the Gramada tool set and how they interact to create a lively workflow. Section 4 describes how this is enabled through adaptations of Ohm/S and the underlying systems. In Section 5, we present the results of our quantitative and qualitative evaluation based on the determined requirements. Section 6 gives an overview over related environments and their approach to providing feedback during language development, and Section 7 concludes the work and points out relevant future work.

2. Elements of Live Programming

An impression of liveness during programming can be achieved through immediate and continuous feedback on the system behavior. Immediate feedback is available when there is only a minimal delay between a modification to the behavior of a system and any feedback on the effects of the change. Continuous feedback is available if any feedback on the behavior of the system is automatically and consistently updated whenever behavior is modified.

The resulting experience of editing a system while it is running has several benefits. In general, it may bridge the "gulf of evaluation" in understanding the relation between the dynamic behavior of a system and its static representation in source code [21]. In particular, some authors argue that humans can only understand causality without conscious effort if there is a minimal temporal distance between their actions and the resulting effects. Thus, a system which enables a short delay for changing the behavior of a software system might give programmers greater insights into the effects of their modifications [17]. Others have shown that novice learners progress faster on tasks when they can work in a live programming environment. Some studies also presented results hinting that novice learners produce significantly less semantic errors while working in live environments, and that in some situations live programming can help in spotting and repairing defects [10, 36]. Finally, live

systems benefit from the improved user productivity and user satisfaction as a result of short response times which have been shown by studies on system response times in various applications [1, 3, 16, 24, 31].

All these benefits make live programming a generally appealing idea. Nevertheless, there are various definitions of the properties defining a live programming system. We combine two perspectives for this work. One regarding the response time between source code edits and modifications of the running application and another regarding the preservation of the context through a "steady frame".

2.1 Immediacy

In order to create an experience of live programming, the programming system should provide *immediate feedback* on how the behavior of the application changes on changes to the source code.

One perspective on live programming distinguishes different levels of live programming based on the temporal relation between an edit and changed behavior [30]. Level four liveness describes systems with short feedback loops: "..., the computer wouldn't wait but would keep running the program, modifying the behavior as specified by the programmer as soon as changes were made." [30] This definition explicitly states that the change should be applied "as soon as" the changes to the source were made and without the developer initiating it explicitly. Further, they should happen while the program is already running. For this criterion, it is irrelevant whether the application keeps on running or whether it is repeatedly executed in a loop because it terminates quickly.

However, these criterion do not explicitly define a threshold for "as soon as". Thus, we based our design and evaluation on two aggregated response time guidelines [12, 25]. To investigate to which degree a development system can provide immediate feedback, we developed a model which distinguishes between factors the system can influence and factors which depend on the particular application under development.

2.1.1 Response Time Guidelines

Shneiderman et al. recommend quantitative upper bounds for the system response time of tasks of different complexity [25]. Resulting from a survey of existing empirical studies, the recommended response times to avoid a frustrating delay are:

- Simple, frequent tasks: 1 second
- Common tasks: 2-4 second
- Complex tasks: 8-12 second

If we want to achieve immediate feedback on modifications to source code, we would need to bring the feedback within the bounds of a frequent task, i.e. less than 1 second.

Johnson takes a different approach to creating quantitative bounds for the system response time. He surveyed existing empirical data on psychological studies on the cognitive capabilities of humans [12]. Two boundaries are relevant for the duration of feedback in programming.

First, the survey concludes that 100 milliseconds (ms) is the maximum delay to sustain the impression of a relation between effect and cause. When the system takes longer than that, it should indicate that it is actually processing the input. Second, 1 second is the maximum time before a system should display a progress indicator, showing how the system is progressing. Otherwise, users might get impatient. This is based on the maximum expected gap in conversations. If this gap is exceeded, listeners turn their attention to the speaker to see what caused the delay. As the interaction with a system is a kind of conversation, this time should not be exceeded.

In order to make observed changes in the behavior of the application feel like effects of the changes to the source code, the system would need to respond within 100 ms. In case the change takes longer, the system should at least indicate that it is processing the change.

2.1.2 Phases of Feedback in Programming

A system provides feedback on various aspects of software development, e.g. editing code, refactorings, or versioning. In this paper, we focus on feedback for a central activity of programming, which is changing the behavior of a system. In particular, we focus on the time span between a modification of the source code and an observable change of the behavior of the system.

We can further distinguish two phases in this time span (see Figure 2): *adaptation* and *emergence*. First, there is the adaptation phase from the completion of a modification of code to a change in the executable form of the system. In most languages or environments, this phase comprises some kind of compilation. Some environments do also support changing the executable form while it is running. Second, there is the emergence phase between the change in the executable form and an observable change in the behavior of the system. This phase is vital to the impression of continuous feedback. However, this phase requires that the changed code is indeed executed. This means that the duration of the emergence phase either depends on the execution behavior of the application at hand or on the execution time of a complete test suite.

These two phases determine the overall time until programmers get feedback on their changes and thereby also the degree of liveness the programmers experience.

2.2 Continuous Feedback

The experience of editing a live system also depends largely on the preservation of context on changes to the application. The context can be preserved through *continuous feedback*, for example, by keeping currently running code executing or

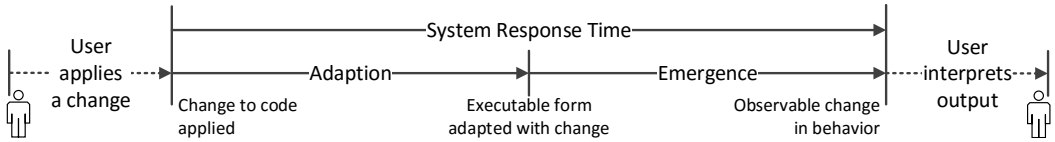

Figure 2. The phases from a modification of the code to an observable change in the behavior, which provides developers with feedback on their changes (adapted from [25]).

preserving and updating the state of interactive tools such as debuggers or read-eval-print loops (REPLs).

The concept of a *steady frame* describes a way to provide live and continuous feedback which preserves the context of an activity [9]. The goal is to make programming a continuous activity such as aiming with a water hose, instead of an activity with discrete independent steps such as aiming through shooting single arrows. An activity with a steady frame is organized such that "(i) relevant variables can be seen and/or manipulated at specific locations within the scene (the framing part), and (ii) these variables are defined and presented so as to be constantly present and constantly meaningful (the steady part)." [9] Therein, the selection of relevant variables mainly depends on the activity and the concrete task at hand. Further, the presentation of the variables also depends on the domain but should make relationships between the manipulated parts of the system and the goal easy to perceive.

Presenting variables in a way so that they are continuously meaningful can become a challenge as program execution often happens in discrete steps. One way to achieve this nonetheless is by structuring the system in terms of data-flow concepts or generally declarative abstractions [9]. These declarative descriptions are constantly in execution and as long as there is real or exemplary data their results can be shown to programmers. For systems based on an imperative execution model, *probes* could be used to achieve a similar effect by continuously watching and displaying the values of relevant variables on exemplary function calls [19].

3. Gramada: Live Language Development

Gramada provides tools to shorten the emergence phase and to provide a steady frame for defining and modifying DSLs. Underlying these tools, the syntax and semantics of the languages are defined through the Smalltalk port of Ohm [4, 35] named Ohm/S which is optimized for short response times (for details see Section 4). In this section we will give an overview over the tools Gramada provides.

Running Example. We will illustrate the features of through a subset of the Questionnaire Language (QL). This language was an assignment of the language workbench challenge 2013[1]. As the challenge aims to allow the comparison of different environments for language development, there are implementations and tutorials for a variety of tool sets. By using QL, we want to enable comparisons between the workflow of Gramada and the workflow of other environments. The QL describes questionnaires which should be rendered from their description. We will implement a subset of the original language, which allows the user to define form fields. As Listing 1 shows, a form field definition contains an identifier, a label presented to the user, and a data type, which is used to render the right form input field. Additionally, Listing 3 shows a small section of a semantics definition rendering a QL form as an HTML form.

Listing 1. An example questionnaire defining two form fields. This is a basic version of the form defined in the language workbench challenge assignment.

```
form Box1HouseOwning {
  hasBoughtHouse: "Have you bought a
    house in 2010?" boolean
  spentOnMaint: "How much have you spent
    on maintenance?" money
}
```

Listing 2. The grammar of the QL written in the Ohm grammar description language.

```
QLQuestionnaire <: BaseGrammar {
  Form = 'form' formName '{' FormBody '}'
  FormBody = QuestionLine*
  QuestionLine = questionIdentifier ':'
    questionLabel typeIdentifier

  formName = letter alnum*
  questionIdentifier = letter alnum*
  questionLabel = '"' (~'"' _)* '"'
  typeIdentifier = letter+
}
```

Tools for Continuous Feedback during DSL Development. Gramada provides various tools: a grammar browser, an input sandbox, a parse tree visualizer, an interactive debugger, and an extension to SUnit [2] for testing syntax definitions. Figure 3 and Figure 4 show how the elementary tools inter-

[1] `http://www.languageworkbenches.net/` (Accessed September 28, 2016)

[2] SUnit is the Smalltalk version of a XUnit testing framework.

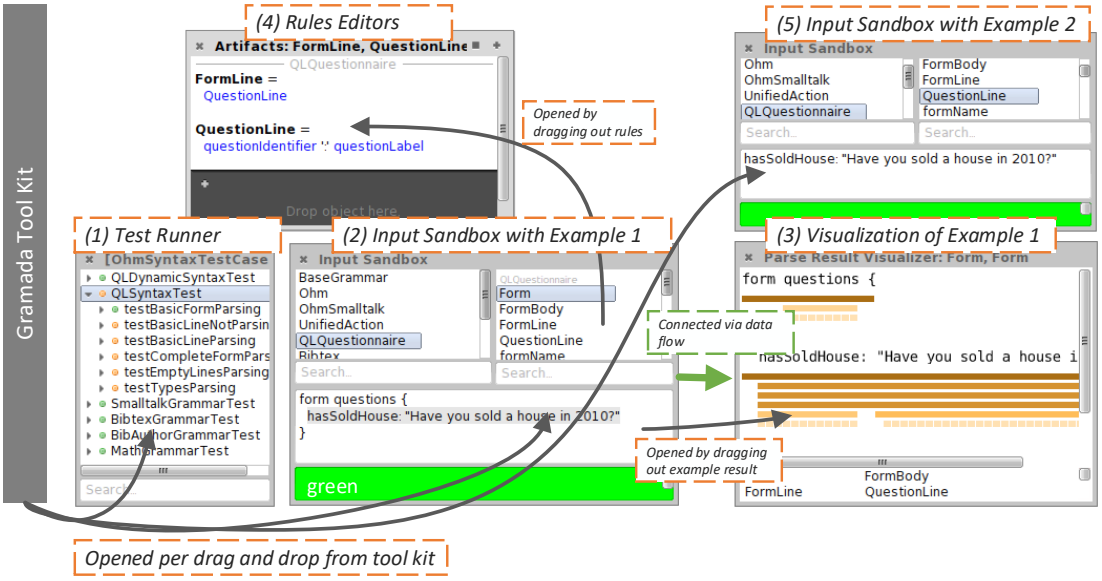

Figure 3. Gramada tools for a programming session to add the type field to the form field description. First, we opened the test runner (1) to get an overview of the failing tests. Then, we opened an input sandbox (2) to explore the current parse tree resulting from a valid QL form. To get a visual impression of the parse tree, we connected the result of the input sandbox to a parse tree visualizer (3). To start our change, we also dragged out interesting rules directly from the rules list (4) and configured another input sandbox directly on the QuestionLine rule (5).

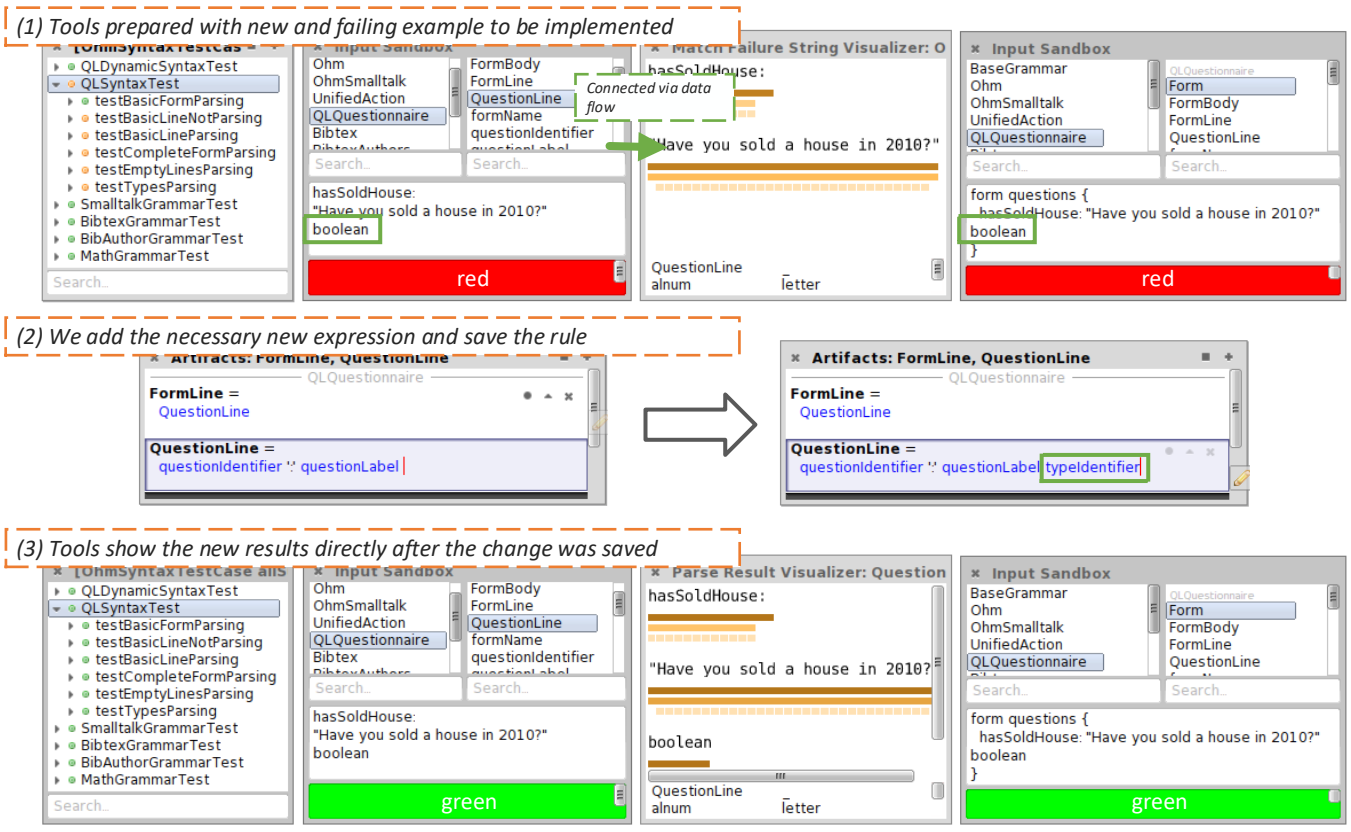

Figure 4. The feedback provided on changes to the grammar. We first modified the examples to include the type specification for a form field which lets the examples fail (1). We then changed the QuestionLine rule to include the type identifier (2). Directly after we saved the rule with a keyboard shortcut, the syntax test runner updates and all the input sandboxes re-evaluate the selected rule on the example. Through a dataflow connection the parse result visualization is also updated (3).

Listing 3. A Smalltalk method defining a semantic of QL which creates an HTML form from the parse nodes resulting from the Form rule of the grammar in Listing 2.

```
Form: aNode with: x FormName: formName
    with: xx FormBody: FormBody
    with: xxx
        ↑ '<form>' , (self value: FormBody)
            , '</form>'
```

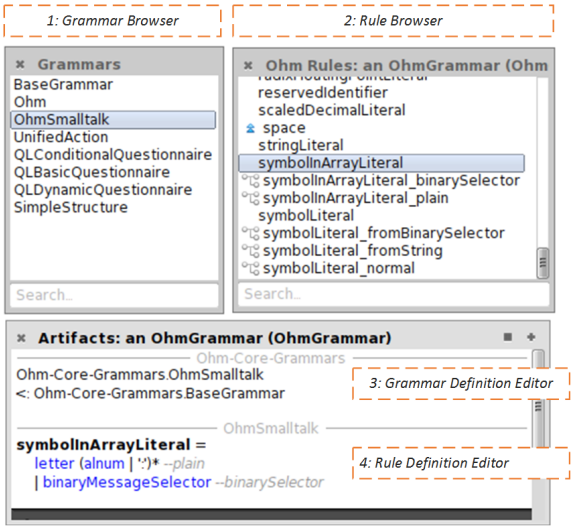

Figure 5. The four basic tools of Gramada: the grammar browser, the rule browser, and the rule and grammar editor both in the same Vivide artifacts pane.

act in a typical programming session for extending the QL language. [3]

3.1 Basic Editing

Gramada provides the developer with dedicated tool support for browsing and editing grammars and rules. As semantics definitions are Smalltalk code, they can be edited using the standard Smalltalk tool set.

The *grammar browser* lists all available grammars in the system (see Figure 5). The *grammar definition editor* allows immediate changes to the grammar definition. Dragging out a grammar from the grammar browser and dropping it outside of a window will open a *rule browser*. It shows all rules of one grammar and denotes their type with symbols for override, extent, and inline rules. The *rule editor* allows the developer to change rules incrementally. Every time the developer saves a change in a rule definition, the rule is compiled and thereby the language implementation changes instantly. If there is a syntax error in the description of the rule, the editor points out the error and does not continue to

[3] See `https://vimeo.com/180190846`. for another demonstration of the Gramada tool set

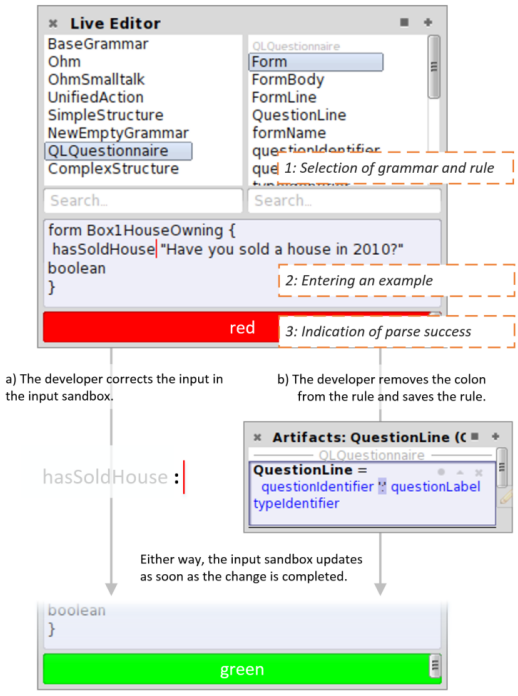

Figure 6. The input sandbox is instantaneously updated on changes to the sample input and on modifications to the currently selected rule.

update the compiled form of the grammar. If several rule editors are showing the same rule and it is modified, then all editors display the new version.

3.2 Input Sandbox for the Exploration of Inputs

When developing a new syntax, one way to see a change in the language, is to see whether the language can now match new inputs. Thus, Gramada provides a steady frame for modifications to a grammar through the *input sandbox tool*, which provides live feedback on the impact of a change on a specific input (see Figure 6). Programmers first choose a grammar and a start rule. Then they can enter the test input and while they are typing the sandbox immediately indicates whether the grammar currently matches the input through the colored bar at the bottom. The feedback becomes relevant within the steady frame as the programmer can choose an example including the syntax elements to be added. The input sandbox does also continuously provide feedback, as whenever any grammar changes, the sandbox will re-evaluate the rule on the input and signal any change (see Figure 6). The selection of a grammar and a rule, as well as the example input, remain stable even when new rules or grammars are added. Further, the input sandbox provides a parse result object which can be passed on to other Gramada tools via a dataflow connection.

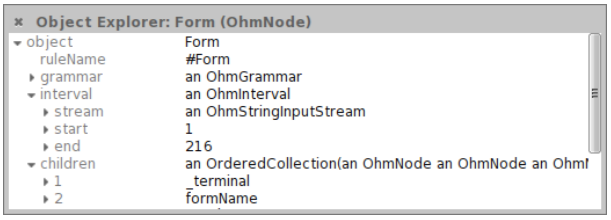

Figure 7. The Vivide object explorer showing a parse result. The tree structure is not obvious and the matched interval is only displayed as the numeric values (start, end).

3.3 Parse Result Visualizer

While the result of the input sandbox is relevant and continuously available, it only shows whether the rule matches the input or not and thus the presentation might not be meaningful. Often, the resulting parse tree structure is also relevant. To explore it, the developer can open the Gramada *parse result visualization*. The result of a parse has three aspects: matched intervals of the input, matched rules, and the resulting tree structure. Smalltalk allows the developer to browse the parse result object structure with an object explorer, but the inherent relationship between the three aspects is not obvious (see Figure 7).

The parse result visualization accepts parse results as input (e.g. from the input sandbox) and visualizes the parse tree structure as a tree in relation to the input string (see Figure 8). This visualization should ease the navigation in the potentially complex structures and make the relation between input, rules, and the parse tree clear. The input is displayed as it was originally entered including line breaks. The matched intervals are displayed as lines underneath the input. When hovering over such an interval line, the name of the originating rule is displayed next to the cursor. The lines are below the input and are ordered from long intervals to shorter ones, as this guides the developer along the common reading direction from the top to the bottom, from an overview to the details. The length is also denoted by color, with longer intervals having a darker color. As for long inputs and complex grammars the parse tree might become too deep to allow a quick overview, the bottom of the visualization includes a list of all involved rules, which allows the developer to filter.

When the parse result visualizer is connected to a live editor, the visualization is also continuously updated on every change to the sample input or on valid changes to the currently used grammar. In case the input can not be matched anymore, the visualizer shows the one complete parse tree which matches the longest section of the input. However, currently Ohm/S does not support incremental parsing and thus this partial result visualization only works with rules containing a many rule, for example FormBody. This is consistent with the way Ohm creates syntax error messages.

Listing 4. An example Ohm/S syntax test cases for the QL. It tests whether the parse returns the correct structure.

```
testBasicFormParsing
  startRule := #Form.
  self shouldParse: 'form␣test␣{}' to:
    #(Form #(formName 'test')
           #(FormBody #()) )
```

3.4 Syntax Testing

When editing an existing grammar, it might be of interest, whether the modification of a rule also changes the behavior of other rules. Automated testing is one way to get fast feedback on changes to the overall behavior of a language implementation. They can also reduce the emergence phase, as a change in the test results is an observable change in the behavior of the language. Further, tests can constitute a general steady frame for the activity of modifying a grammar which provides grammar tests. The test results are relevant to programmers as they indicate whether the other rules still work as expected. Also, the *syntax test browser* continuously displays the complete result. Therefore, Gramada includes an extension to SUnit to ease writing tests for syntax definitions. The extension is based on the Spoofax testing language SPT [14].

The definition of *syntax tests* is based on XUnit concepts. To test parsing results, the test case class provides two methods: one for testing whether the input was matched at all, and one for testing whether the parse result matches a given structure (see Listing 4).

Through the integration with SUnit, the syntax tests integrate with SUnit tools, like the Squeak auto tester tool [26] which automatically executes tests whenever the tested package changes. The Squeak auto tester tool currently requires users to manually select the package containing the code to be observed and the package containing the tests to be executed. A specialized syntax test runner could use the information on the tested grammar and rule in the syntax tests for a more specific test case selection. To integrate the results of syntax test cases with other Gramada tools, the tool set also includes a *syntax test browser* (see Figure 8). For each test case it displays the parse results generated throughout the execution of the test. Finally, the test results can be dragged out from the syntax test browser by the language developer to open up a debugger or an input sandbox on them.

3.5 Grammar Debugger

To aid language developers to explore parsing failures and trace them to the defects in the grammar rules, Gramada includes an interactive *grammar debugger*. In its design it is similar to the attribute grammar debugger Aki [23]. For novice language developers, the debugger might ease the

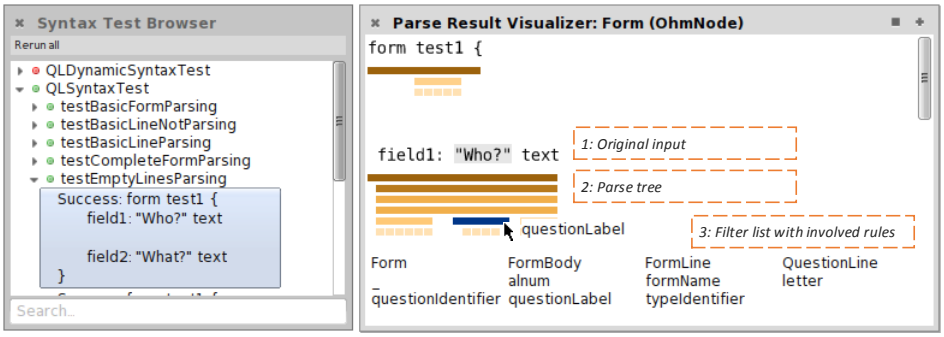

Figure 8. The Gramada syntax test runner and the parse result visualizer. The visualizer shows the parse of the example questionnaire. The tree structure is represented through stacked lines which reveal detailed information like matched input interval and rule name on mouse hover.

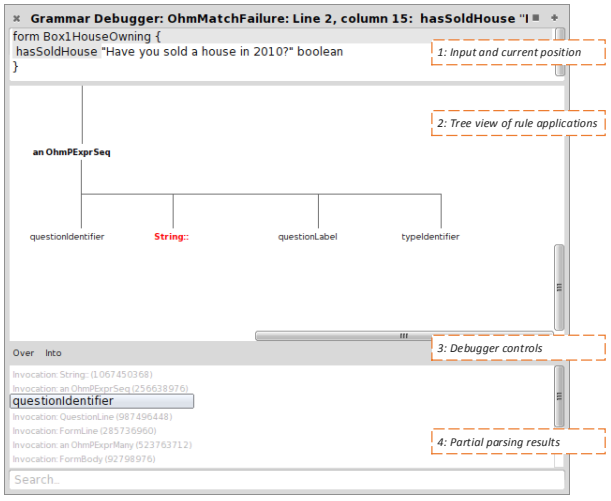

Figure 9. The graphical Gramada debugger, debugging a missing colon in an example questionnaire.

process of understanding their grammars without requiring up-front knowledge of the concrete parsing algorithm. This requirement for in-depth knowledge of the parsing algorithm is often a hurdle as the assumptions of a concrete parsing algorithm often differ from the general idea of a formal grammar, for example regarding left-recursion. Instead of learning the abstract parsing algorithm by heart, they can interactively explore the resulting dynamic behavior in the context of an example relevant to them.

The debugger allows the developer to interactively step *into* or *over* a rule application. It shows the progress in the input string and the position in the rule application stack in the form of a graphical tree. In case a rule has several subexpressions, the debugger also shows partial parsing results in the list at the bottom. These list items can be inspected with the parse result visualizer or used as the input for interactive tools. The debugger is configured by connecting the result of an input sandbox. At this point, the debugger does not yet provide a steady frame presentation of an execution. Changes to the grammar will not affect the current debugging progress. It will neither update the future rule invocations nor replay the debugging session using the modified rule behavior.

3.6 Connecting the Tools to a Workflow

As the tools are all built on top of Vivide, several of them interconnect. Whether two tools can be connected depends on what type of Ohm/S object they display.

Through the connections between the tools, navigating between them becomes easier. When an issue arises, the developer can open new tools to understand the issue directly from within the tool initially bringing up the situation. For example, the developer might change a rule and the moment the rule is saved the *syntax test browser* indicates that a syntax test just broke. The developer then drags out the faulty parse result which opens a *parse result visualizer* on the object that shows how much of the input was matched. To drill down on the problem, the user connects the visualizer to an *input sandbox* to automatically set its grammar, start rule, and input. In the editor, the developer reduces the input to a minimal example causing the parsing failure. To finally figure out the issue, the developer connects the result of the input sandbox to a *grammar debugger* and steps through the parsing process to find the defect in the rules. As the appropriate solution is not clear, the developer experiments with several changes to the rules and the opened tools all update continuously after each edit to a rule.

4. Implementation

The quick availability of feedback in Gramada tools is made possible by the underlying language implementation framework Ohm/S. This Smalltalk port of the Ohm framework [4, 33, 35] uses implementation techniques known from general-purpose programming language implementations to achieve a short response time. Further, Gramada profits from the tool building capabilities of the *Vivide* environment.

4.1 Technical Foundations of Gramada

Gramada is based on previous work for defining languages and creating tools. We give a short overview on the technical foundations to distinguish between their features and the technical contributions of Gramada.

4.1.1 Language Definition Framework Ohm

Ohm is a language definition framework based on Parsing Expression Grammars (PEGs) that supports a novel form of modular semantics definitions. This allows for a clean separation of syntax and semantics, with a small and statically checkable interface between them. Ohm syntax definitions are written in an external DSL, which makes them reusable across different implementations of Ohm. For example, Listing 2 shows an Ohm grammar accepting the QL. The grammars are interpreted as PEGs with left-recursion support [34]. Further, the grammars support various rule inheritance semantics (inherit, extend, and override) from exactly one supergrammar. The semantics of a language are defined through abstractions of the host language [35], e.g. objects in Javascript and classes in Smalltalk.

4.1.2 Tool Development Environment Vivide

Gramada implements a set of tools which are all based on and integrated into Vivide, an environment which eases the development of graphical tools by taking a data-driven perspective on them [29]. As a framework, it helps developers to develop graphical tools by separating the code which prepares the data to be displayed from the code which deals with the details of graphical components, like rendering or event handling. As a programming environment, it changes the workflow, as the objects themselves become central to user interactions. For example, instead of opening a class browser, the developer opens a set of class objects, optionally filters and transforms them, and finally selects a graphical component to display the resulting data set. It also allows the programmer to combine existing tools by defining a dataflow between them, such as the connection between the input sandbox and the parse result visualizer.

Scripts and Views In Vivide, the code which prepares the data for displaying is represented as a *script*. The visual component displaying this data is called a *view* and can be a list, a tree, or even a three dimensional visualization of a graph. Typically, one tool has several scripts which transform domain objects to the target data set and the corresponding view configurations. A view will get the resulting view configuration as an input and render the elements appropriately. For example, if the collection of methods is rendered as a list, then each method becomes a list item with the method selector as its label. The combination of scripts and a view is captured in a *pane*, for example the rule browser combining a script for extracting rules from a grammar and a list view.

Dataflow Every pane has an output which is determined by the view. For example, when a user selects a method from a list view containing methods, the output of the pane of the list is the selected method object. This output can be connected to the input of another pane. For example, a developer could create a list of classes and connect its output to the input of our pane, listing the methods of a class. These combinations of panes through connections can also be grouped together to form single tools.

4.2 Ohm/S

To define the syntax and semantics of DSLs, Gramada uses Ohm/S, which is a Smalltalk implementation of the language development framework Ohm. Thus, the syntax of a language is defined using the Ohm grammar language (see for example Listing 2) and the semantics are defined through Smalltalk classes and methods.

Ohm/S primary design goal is to provide the programmer-friendly language development features of Ohm while supporting sufficiently short adaptation phase to enable immediate feedback in Gramada. Gramada should evoke the impression that the developer is editing the grammar itself and not a character string. Whenever the code of a rule changes, the grammar behavior should change directly and any input matching in progress should be quickly continued or restarted. Thus, in order to decrease the adaptation time in the domain of language development, Ohm/S uses the same kind of engineering techniques which enable immediacy for application development.

4.2.1 Incremental Compilation

Ohm/S enables incremental compilation through its metamodel. Similar to Smalltalk classes, grammars are common objects in the Smalltalk image, protected from the garbage collector by a global reference. Each grammar contains a dictionary mapping rule names to rule objects. To change a rule, it is sufficient to translate this one rule and to replace it in the rules dictionary of the corresponding grammar. Changes to the grammar definition are interpreted as changes to the state of the grammar object itself. Hence, changing the supergrammar does not require a complete translation of the grammar but does only change the supergrammar instance variable of the existing grammar object. Thus, the translation of rules and grammars is spread throughout the editing. As a result, an updated grammar can always be used directly without any further translation.

4.2.2 Ohm/S Dynamic Inheritance

In Ohm/JS, a grammar inherits all rules of its supergrammar when it is compiled from its definition. If an inherited rule changes in the supergrammar at a later point in time, the child grammar will not be updated, but keep the old inherited behavior. As Ohm/S grammars are persisted and seldom recreated from their definition, we changed the inheritance semantics to dynamic inheritance in Ohm/S. Whenever a grammar tries to apply a rule which it can not find in its rule dictionary, it asks a supergrammar for the rule. This

means that the definition of a rule can now change through modifications in the grammar itself or through modifications in the supergrammar. Additionally, if the rule specifies a particular inheritance semantic, e.g. the child extends the parent rule with another alternative, the inheritance operator is applied during the lookup, e.g. by combining the parent rule and the child rule to form a new rule. The results of the lookup are cached for the duration of one parsing process.

4.2.3 Defining Semantics

In Ohm/S, semantics are defined in classes. A language developer can subclass from a class which represents how the semantics definition should be interpreted. The single functions for parse tree nodes are defined in methods. The default attributes are provided as methods in the semantics superclasses. The mapping between the parse nodes of a particular rule and the function computing the corresponding attribute value is established on a syntactical level, so the method name has to match the rule name in a certain way. An attribute can be evaluated recursively by calling value on itself, passing the next nodes to be evaluated (see Listing 3).

5. Evaluation

Gramada is designed to provide immediate and continuous feedback for the development of DSLs. We quantitatively evaluated Gramada regarding the goal of immediate feedback. We also qualitatively discuss the capabilities of Gramada to provide continuous feedback through a steady frame for changing a grammar.

5.1 Evaluation of the Adaptation Phase

For the evaluation, we focus on the feedback for developing the syntax of a language. Gramada does not include features designed to shorten the phases of semantics. However, as the developer specifies semantics through ordinary Smalltalk methods and classes the short adaptation phase durations of Smalltalk systems holds.

5.1.1 Adaptation Time of Syntax Definitions

To determine the adaptation time, we conducted an experiment based on the 113 Ohm/S grammar rules in the image. We measured the time between the start of the compilation and the end of the installation of the rule into the grammar. We measured every translation 100 times. During every run, the garbage collector was deactivated. As the lines of code of a rule do not correlate with its syntactic complexity, we used the number of expression nodes in the compiled form of a rule as the parameter of our benchmark. We used the following setup to conduct the experiment:

- Intel i5-3320M with a 2.6 GHz frequency and 8 GB DRAM running Windows 7 Enterprise 64 Bit
- Squeak 4.5 (Update 13680) with a Cog VM version CoInterpreter VMMaker.oscog-eem.331 (Win32 built on Aug 22 2013 10:20:54 Compiler: 3.4.4)

Table 1. Measured running times in milliseconds for translating a complete grammar (100 runs each).

Grammar	#Rules	Median	Range
BasicQuestionnaire	8	311	311-316
DynamicQuestionnaire	11	488	476-594
Ohm	61	3219	2963-3529
Smalltalk	76	3913	3694-4341

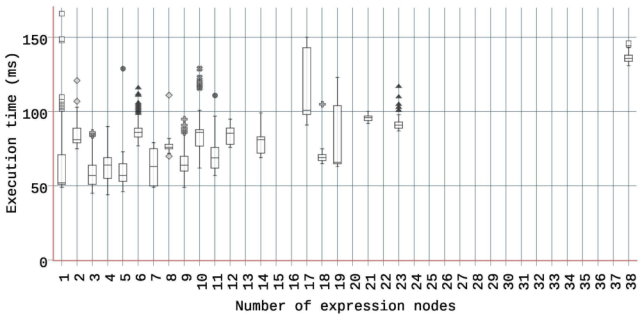

Figure 10. A Tukey box plot of all measurements of adaptation times showing that the median times stay under 100 ms and only one outlier exceeds 150 ms. The fact that the times only slightly increase with the growing complexity of a rule stems from the fact that the serialization phase does mainly depend on the size of the grammar and not on the size of the rule (for details see Appendix A). This impact could be mitigated through caching.

- Additional packages loaded: Ohm−Core, Ohm−Grammars (From `https://github.com/hpi-swa/Ohm-S`) with the commit hash `5f6d09e9`

5.1.2 Discussion of Results

For the adaptation phase of syntax definitions, we have found that the median adaptation time stays below 100 ms for rules with less than 24 expression nodes and otherwise stays below 150 ms (see Figure 10). Further, a comparison to a complete recompilation of grammars shows, that for Gramada, incremental compilation is key to achieving short adaptation times. Translating a complete grammar after each change to the source code is only feasible for small grammars (see Table 1). For larger grammars, the adaptation time is already within Shneidermans recommended range of common tasks (2-4 seconds) and Johnsons range of durations which make users impatient.

5.2 Evaluation of the Emergence Phase

The emergence delay is the time between a change in the executable form of a language and an observable change in the behavior of the language implementation. To observe a change in behavior, we have to re-execute the implementation after each change. For syntax definitions, Gramada

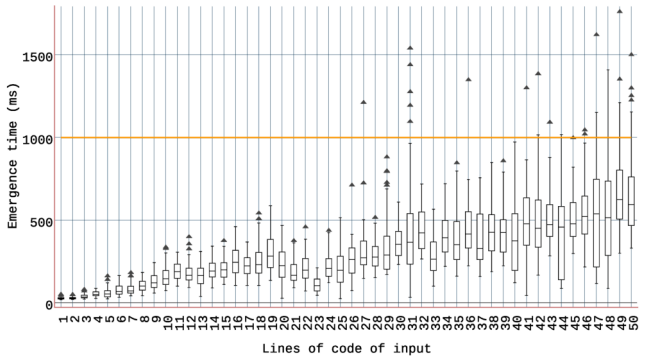

Figure 11. A Tukey box plot of measurements of the emergence phase of the input sandbox connected to the parse tree visualizer. This plot leaves out 7 outliers with execution times of over 2 seconds. Most median times stay under 500 ms. The increase in variance stems from an increasing variance in method complexity with increasing method length. Overall, there is a linear increase in the median execution time (Pearson coefficient: 0.958).

achieves this through the input sandbox and testing abstractions enabling auto-testing.

The semantic definitions generally profit from the features shortening the emergence phase in Squeak/Smalltalk, such as the dynamic debugger, the object explorers, or the hot-swapping support. Beyond that, Gramada does not provide tool support to provide a tailored live programming experience for the definition of semantics (see Section 7).

5.2.1 Emergence Time of the Input Sandbox

The input sandbox of Gramada allows direct observations of changes in the behavior of the parser of a language implementation. Optionally, the input sandbox can also update the parse result visualizer on every keystroke. As this combination provides the most information on changes to the source code of a rule, i.e. whether the rule matches and what the complete parse tree looks like, we decided to evaluate the emergence time of the two in combination.

5.2.2 Experiment Setup

We assume that the response time of the input sandbox depends mainly on the selected grammar and the provided input. We have chosen the Ohm/S Smalltalk grammar as a representative input grammar for a general-purpose programming language. We assume that DSLs will reach a similar, but not a higher degree of syntactic complexity. We vary the input complexity by randomly sampling 250 Smalltalk methods, five for each size between 1 to 50 lines. We have chosen lines of code as the benchmark parameter, as we are interested in an overall trend and upper bounds of execution times with regard to realistic input and not in the particular factors making up the response time. Every method is entered into the input sandbox 10 times. We measure the time from the completion of the insertion up to the end of the rendering of the parse result tree. The benchmark setup consists of the hardware described in Section 5.1.1 with the following changes to the software components:

- Squeak 4.5 (Update 15059) with a Cog VM (Win32 built on Jun 27 2015 12:57:31 PDT Compiler: 3.4.4)
- Gramada revision ab493a4f-1a88-1e45-a081-2457fc3c5d34

5.2.3 Auto-Testing for Syntax Definitions

The automatic execution of tests provides feedback on the overall behavior of a grammar. The resulting response time depends mainly on the number of test cases. So far, the Gramada syntax test runner is based on the Squeak auto tester tool which naively executes all tests in a package on a change. To determine, whether this strategy is sufficient for a live programming experience, we measured the execution time of 100 runs of the Smalltalk grammar tests which reimplement all syntax tests the original parser package provided which results in 34 test methods, each with up to 15 input/rule combinations. This test class had a median execution time of 1520 ms (standard deviation: 329.47 ms). As the tests always run concurrently this is bearable. However, the threshold for making users impatient (1 second) is exceeded. A more advanced test case selection would be required to scale the auto-testing setup to larger test sets.

5.2.4 Discussion of Results

To assess the emergence phase, we analyzed the evaluation delay of the input sandbox. For short inputs, the delay of the input sandbox stays under 500 ms and for larger ones the median delay remains under 1 second (see Figure 11). To assess the overall duration of the feedback, we have to examine the combination of the adaptation and the emergence phase. Using the input sandbox, the overall duration until feedback is available will stay under 1 second for short inputs and might occasionally exceed this threshold for longer inputs.

According to Shneiderman, this makes the input sandbox fast enough to be used frequently. The auto tester tool is fast enough to be a commonly used tool. Regarding Johnson's thresholds, the overall system response time of all the evaluated tools exceeds the threshold for the impression of causation. Only for very short inputs (<5 lines of code for Smalltalk methods) the delay between changing the input and seeing a change in the parse tree visualization stays under this threshold. Nevertheless, most scenarios stay under the threshold for making users impatient.

5.3 Continuous Feedback in Gramada

Gramada aims to provide continuous feedback through creating a steady frame for the activity of changing a grammar. The tools therefore need to provide a frame for the presentation and modification of relevant variables as well as a steadily meaningful presentation of these variables [9].

Framing is achieved by the tools themselves. Each tool shows a different part of the system, for example, the live

editor shows whether a certain input can match at all, and the syntax test browser shows the results of all syntax tests in the system. The relevancy of these parts depends on the task at hand, for quick feedback the live editor is sufficient, for an in-depth exploration the parse result visualizer presents the parse tree as a relevant part.

While the framing is a direct result from the distinct tools, the steadiness is a result of the way the Gramada tool set and the Ohm/S framework interact. In comparison to dataflow-defined concepts, grammars are not constantly running. So changes in the behavior of a grammar would only manifest as soon as the grammar is explicitly used with concrete examples. To make the presentation of feedback such as the parse result visualizer constantly meaningful, the Gramada notification system triggers an update in all corresponding tools currently showing aspects of the modified grammar. To keep the feedback steady, the selection of grammars, rules, or examples is not changed on changes to the way the grammar works.

6. Related Work

The development of domain-specific languages has been subject to extensive research. Thus, there are a variety of solutions for supporting DSL developers, ranging from parsing libraries and corresponding tools to complete environments for language-oriented programming, which include language and application development facilities.

6.1 General Language Development Environments

Gramada focuses on tool support for immediate feedback during language development. Therefore, in our discussion of general language development solutions, we focus on approaches providing mechanisms for short feedback loops. One of them is *ANTLRWorks* which provides interactive tooling for the grammar language *ANTLR*. Another environment called *PetitParser* achieves short system response times for its syntax definitions through using an internal DSL based on Smalltalk.

ANTLRWorks. ANTLRWorks is a development environment for ANTLR, a framework for compiler and interpreter construction [2]. In ANTLR, a language is implemented with grammars which define the syntax and tree parsers which define the code generation for a target language. The basic ANTLR workflow contains an explicit generation step from a set of grammars to source code describing the DSL, e.g. the QL, in a target language, e.g. Java.

ANTLRWorks contains an ANTLR grammar interpreter which allows using the language specifications without translating the grammars first. Based on this interpretation, ANTLRWorks provides a static expression graph visualization and a parse tree visualization for example input strings. As the grammars are interpreted, there is no adaptation step involved. However, the parse tree visualization has to be refreshed manually. Another consequence of the grammar interpreter is that the semantics definitions do not work semantically correct and can not be evaluated as quickly as the syntax definitions.

PetitParser. PetitParser is a parser combinator library written in Smalltalk [22]. It can be used to specify parsers but not semantics. PetitParser also provides a tool set, which includes a static expression graph of a parser, a view showing the parsing progress through an example input which includes a backtracking visualization, a parsing profiler, and a debugger that allows navigating a static trace. Changes to the parser definition are updated by saving the containing Smalltalk method. All the described tools are updated on saving the example input but not on changes to the parser. Thus, the feedback on the impact of changes is delayed as the examples have to be manually refreshed.

PetitParser does provide immediate feedback on changes to the language implementation as the syntax definitions are implemented in an internal Smalltalk DSL. Thus, PetitParser benefits from the Smalltalk tooling and short adaptation times.

6.2 Language Workbenches

Language workbenches are environments designed to simplify developing new languages and the corresponding development environments. At the time of writing, there are numerous language workbenches developed and maintained [5]. We considered the particularly prominent and advanced environments Spoofax, Xtext, and Rascal.

Spoofax. Spoofax is a mature language workbench for specifying languages and corresponding development environments [13]. It integrates a number of specialized languages for language development, e.g. SDF3 for syntax definitions based on context-free grammars and the Stratego transformation language. Spoofax provides an editor with code navigation between specification languages, an outline for SDF3 grammars, a visualization of a parse tree of an example input, and a testing language for syntax and semantics definitions including a test runner.

The translation of language definitions to an implementation is initiated manually, and takes several seconds, excluding the duration for generating the new tools. Afterwards, the developer has to manually refresh any opened input visualizations or test runners. Spoofax does not include an interactive debugger. However, due to the design of the SDF3 syntax definition formalism, some programming errors possible in PEG-based grammar languages can not occur, e.g. wrong ordering of choices.

Xtext. The Xtext workbench allows syntax definitions through grammars and semantics definitions through code generation. Based on the syntax definition, Xtext can generate an editor, including code highlighting and navigation for the language under development [6]. Xtext also provides an editor with code highlighting and navigation for

the syntax definitions. As the code generators are defined in Java, the DSL developer can use all Java tools provided by Eclipse. Besides the editor, Xtext does currently not provide any other tools for the language developer. Also, to use the newly developed language, the developer has to explicitly start a generation process and launch a new Eclipse instance. The complete process can take several seconds.

Rascal. At the core of the Rascal environment is the Rascal programming language, which is a general-purpose programming language with extensions for programming language specification [15].

In contrast to other language workbenches, Rascal provides tooling mostly through an interactive REPL integrated into Eclipse. This REPL allows the developer to dynamically load language specifications like modules. If the language exposes an eval or parse function, then it can be used interactively in the Rascal REPL. Further, the command line provides a module to visualize a parse result as an interactive parse tree. The REPL allows interactive exploration of a language implementation. However, whenever the language definition changes, the developer has to restart the language console manually to use the changed definition. Currently, Gramada does not provide special tools for the interactive exploration of semantics yet. Instead, the Squeak/Smalltalk tools, such as the workspace and the object explorer, can be used to explore the language.

7. Conclusion and Future Work

Live programming based on immediate and continuous feedback is beneficial for program comprehension and programmer productivity. Especially, feedback on the impact of changes to the system is important. With regard to immediacy of feedback, we distinguished the adaptation and the emergence phase. We further used the previously defined concept of steady frames to qualify continuous feedback.

Gramada We proposed the language development environment Gramada, designed to provide immediate and continuous feedback on changes to DSLs. First, it includes the language development framework Ohm/S, which is a Smalltalk adaptation of the Ohm [4, 35] framework. Ohm/S additionally features the incremental compilation of grammars and a dynamic lookup of rules. Second, Gramada provides tools, like the input sandbox and the syntax test runner, which create a steady frame for the activity of modifying grammars through continuously and consistently providing feedback on the behavior of grammars.

Squeak/Smalltalk Implementation We implemented Gramada on top of Squeak/Smalltalk and Vivide. The implementation of Ohm/S demonstrates how engineering techniques used to decrease system response times during application development can also be applied to language development. The tools shorten the emergence phase by re-evaluating examples on changes to the examples or the language definition.

Evaluation Results In a quantitative analysis of the response times of Gramada, we have found that the adaptation phase of rules is generally shorter than 200 ms. Also, the input sandbox combined with the parse tree visualizer provides a median emergence phase duration of under 500 ms for the grammar of a general-purpose language. This results in an overall response time of under 1 second. The auto tester tool running syntax tests results in an emergence phase of 1 to 3 seconds. We further discussed properties of the steady frame the Gramada tool set can provide.

These results show that live programming is possible in language development. Thereby, Gramada can improve the overall feedback loop of DSL development and make language implementations easier to understand. This can help making DSLs more accessible in software development.

Future Work. Gramada enables live language development. In addition to improvements on the tool set, it also brings up new research opportunities.

Although short feedback loops and the resulting impression of immediacy are generally beneficial, it remains to be shown how they impact the process of understanding, as well as how they impact developers for the particular task of language development. Both aspects, productivity and accessibility, of language development could be evaluated with respect to changes to the response time in Gramada.

Gramada provides several tools providing immediate and continuous feedback for syntax definitions. So far, the semantics can be developed with short feedback loops, as they are defined through Smalltalk methods. While the workspace in Squeak/Smalltalk allows the interactive exploration of language semantics, it still requires manual interventions to produce results. Additional tool support can help to shorten the emergence phase for the semantics definitions further. To keep response times low despite re-evaluations of examples, an incremental evaluation of semantics would be beneficial, e.g. for the the HTML rendering of a QL form. Instead of re-evaluating the changed attribute on the whole parse tree, the evaluation can be limited to affected sub-trees.

Further, the Gramada tool set could be extended to a language workbench. To achieve this, it would not only require concepts to describe and generate tool support for the described DSLs, but also mechanisms to enable immediate and continuous feedback on changes to them. Gramada should allow the developer to create the DSL tools in the same environment and see changes to them directly after a modification of a definition. Thereby, it could create an environment providing immediacy for developing new languages, for using these languages, and for common general-purpose application development.

Acknowledgments

We gratefully acknowledge the financial support of the Research School of the Hasso Plattner Institute and the Hasso Plattner Design Thinking Research Program. We would like to thank Alessandro Warth for his substantial support and feedback. We also thank Jens Lincke and Richard P. Gabriel for countless fruitful discussions.

References

[1] R. E. Barber and J. Lucas, Henry C. System response time operator productivity, and job satisfaction. *CACM*, 26(11): 972–986, 1983.

[2] J. Bovet and T. Parr. ANTLRWorks: An ANTLR grammar development environment. *Software: Practice and Experience*, 38(12):1305–1332, 2008.

[3] J. Dabrowski and E. V. Munson. 40 years of searching for the best computer system response time. *Interacting with Computers*, 23(5):555–564, 2011.

[4] P. Dubroy, S. Kasibatla, M. Li, M. Röder, and A. Warth. Language hacking in a live programming environment. In *Proceedings of the LIVE Workshop co-located with ECOOP 2016*, 2016. URL `https://ohmlang.github.io/pubs/live2016/`.

[5] S. Erdweg, T. v. d. Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. v. d. Vlist, G. H. Wachsmuth, and J. v. d. Woning. The state of the art in language workbenches. In *Proceedings of SLE 2013*, pages 197–217. Springer, 2013.

[6] M. Eysholdt and H. Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of OOPSLA 2010*, pages 307–309. ACM, 2010.

[7] M. Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[8] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA, 1983.

[9] C. M. Hancock. *Real-Time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Massachusetts Institute of Technology, Sept. 2003.

[10] C. D. Hundhausen and J. L. Brown. What you see is what you code: A live algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing*, 18(1):22 – 47, 2007.

[11] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *ACM SIGPLAN Notices*, volume 32, pages 318–326. ACM, 1997.

[12] J. Johnson. *Designing with the Mind in Mind*. Morgan Kaufmann, San Francisco, CA, USA, 2nd edition, 2014.

[13] L. C. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of OOPSLA 2010*, number 10, pages 444–463. ACM, 2010.

[14] L. C. L. Kats, R. Vermaas, and E. Visser. Integrated language definition testing: Enabling test-driven language development. In *Proceedings of OOPSLA 2011*, pages 139–154. ACM, 2011.

[15] P. Klint, T. van der Storm, and J. Vinju. EASY Meta-Programming With Rascal. In *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering 2009*, volume 6491 of *Lecture Notes in Computer Science*, pages 222 – 289. Springer, 2011.

[16] G. N. Lambert. A comparative study of system response time on program developer productivity. *IBM Systems Journal*, 23 (1):36–43, 1984.

[17] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of SIGCHI 1995*, pages 480–486, New York, New York, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[18] J. McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1965.

[19] S. McDirmid. Usable live programming. In *Proceedings of the Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!) 2013*, Onward! 2013, pages 53–62, New York, NY, USA, 2013. ACM.

[20] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[21] D. A. Norman and S. W. Draper. *User Centered System Design*. Lawrence Erlbaum Associates, Inc., Publishers, 1986.

[22] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz. Practical dynamic grammars for dynamic languages. In *Proceedings of DYLA 2010*. Springer, 2010.

[23] M. Sassa and T. Ookubo. Systematic debugging method for attribute grammar description. *Information Processing Letters*, 62(6):305 – 313, 1997.

[24] W. Scacchi. Understanding software productivity: Towards a knowledge-based approach. *Journal of Software Engineering and Knowledge Engineering*, 1(3):293–321, 1991.

[25] B. Shneiderman, C. Plaisant, M. Cohen, and S. Jacobs. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson, Upper Saddle River, New Jersey, USA, international edition of 5th revised edition, 2009.

[26] Software Architecture Group, Hasso Plattner Institute. Squeak AutoTDD [Computer Software]. Retrieved from `https://github.com/HPI-SWA-Teaching/AutoTDD`. visited September 28, 2016.

[27] Squeak Foundation. Squeak/Smalltalk version 4.6 [Computer Software]. Retrieved from `https://www.squeak.org`. visited September 28, 2016.

[28] M. Taeumel. Vivide [Computer Software]. Retrieved from `https://github.com/hpi-swa/Vivide`. visited September 28, 2016.

[29] M. Taeumel, M. Perscheid, B. Steinert, J. Lincke, and R. Hirschfeld. Interleaving of modification and use in data-driven tool development. In *Proceedings of Onward! 2014*, pages 185–200. ACM, 2014.

[30] S. L. Tanimoto. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming (LIVE) 2013*, pages 31–34. IEEE, 2013.

[31] A. J. Thadhani. Factors affecting programmer productivity during application development. *IBM Systems Journal*, 23(1): 19–35, 1984.

[32] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA 1987*, pages 227–242, New York, New York, USA, 1987. ACM.

[33] A. Warth and P. Dubroy. Ohm version 0.85 [Computer Software]. Retrieved from `https://github.com/cdglabs/ohm`. visited September 28, 2016.

[34] A. Warth, J. R. Douglass, and T. Millstein. Packrat parsers can support left recursion. In *Proceedings of PEPM 2008*, pages 103–110, New York, New York, USA, 2008. ACM.

[35] A. Warth, P. Dubroy, and T. Garnock-Jones. Modular semantic actions. In *Proceedings of DLS 2016*, New York, New York, USA, November 2016. ACM.

[36] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *Proceedings of CHI 1997*, pages 258–265, New York, NY, USA, 1997. ACM.

A. Detailed Benchmark Results

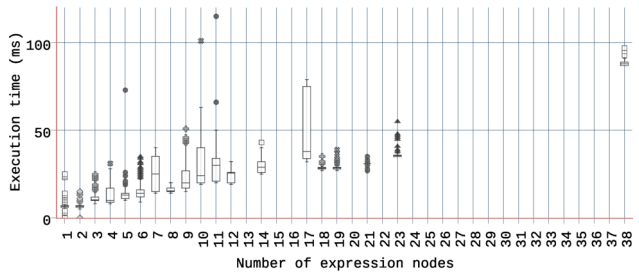

Figure 12. A Tukey box plot of all measurements of adaption times only containing the rule compilation part with regard to the number of expressions nodes in the rule abstract syntax tree. The plot shows a linear increase in compilation time (Pearson correlation coefficient of 0.935).

In addition to the overall adaptation phase duration in Ohm/S we have also measured the duration of the two phases making up the adaptation phase: compiling a rule and serializing a grammar into a Smalltalk class for persistence. When looking at the result diagram (Figure 10) there is a recognizably slow increase of the adaptation phase duration (see Figure 12) with the increase in the node count. The measurements of the two individual phases reveal that this results from the fact, that the serialization phase duration (see Figure 13) does not depend on the rule complexity but makes up a considerable portion of the overall duration.

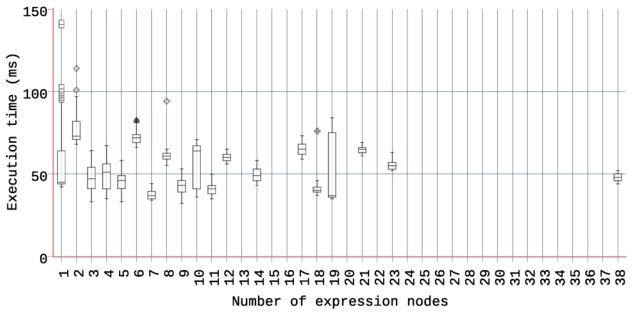

Figure 13. A Tukey box plot of all measurements of adaption times only containing the serialization part. The chart indicates that there is no correlation between the execution time and the rule complexity (Pearson correlation coefficient of -0.093).