

# Squimera: A live, Smalltalk- based IDE for dynamic programming languages

Fabio Niephaus, Tim Felgentreff, Robert Hirschfeld

**Technische Berichte Nr. 120**

des Hasso-Plattner-Instituts für  
Digital Engineering an der Universität Potsdam





Technische Berichte des Hasso-Plattner-Instituts für  
Digital Engineering an der Universität Potsdam



Technische Berichte des Hasso-Plattner-Instituts für  
Digital Engineering an der Universität Potsdam | 120

Fabio Niephaus | Tim Felgentreff | Robert Hirschfeld

## **Squimera**

A live, Smalltalk-based IDE for dynamic programming languages

### **Bibliografische Information der Deutschen Nationalbibliothek**

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.dnb.de/> abrufbar.

### **Universitätsverlag Potsdam 2018**

<http://verlag.ub.uni-potsdam.de/>

Am Neuen Palais 10, 14469 Potsdam

Tel.: +49 (0)331 977 2533 / Fax: 2292

E-Mail: [verlag@uni-potsdam.de](mailto:verlag@uni-potsdam.de)

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam.

ISSN (print) 1613-5652

ISSN (online) 2191-1665

Das Manuskript ist urheberrechtlich geschützt.

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam

URN [urn:nbn:de:kobv:517-opus4-403387](http://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-403387)

<http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-403387>

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:

ISBN 978-3-86956-422-7

# Abstract

Software development tools that work and behave consistently across different programming languages are helpful for developers, because they do not have to familiarize themselves with new tooling whenever they decide to use a new language. Also, being able to combine multiple programming languages in a program increases reusability, as developers do not have to recreate software frameworks and libraries in the language they develop in and can reuse existing software instead.

However, developers often have a broad choice with regard to tools, some of which are designed for only one specific programming language. Various Integrated Development Environments have support for multiple languages, but are usually unable to provide a consistent programming experience due to different features of language runtimes. Furthermore, common mechanisms that allow reuse of software written in other languages usually use the operating system or a network connection as the abstract layer. Tools, however, often cannot support such indirections well and are therefore less useful in debugging scenarios for example.

In this report, we present a novel approach that aims to improve the programming experience with regard to working with multiple high-level programming languages. As part of this approach, we reuse the tools of a Smalltalk programming environment for other languages and build a multi-language virtual execution environment which is able to provide the same runtime capabilities for all languages.

The prototype system `SQUIMERA` is an implementation of our approach and demonstrates that it is possible to reuse development tools, so that they behave in the same way across all supported programming languages. In addition, it provides convenient means to reuse and even mix software libraries and frameworks written in different languages without breaking the debugging experience.

# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
1.1	The Need for Software Development Tools . . . . .	12
1.2	The Need for Software Libraries and Frameworks . . . . .	12
1.3	Contributions . . . . .	13
1.4	Outline . . . . .	14
<b>2</b>	<b>Context</b>	<b>15</b>
2.1	Virtual Execution Environments . . . . .	15
2.2	The Smalltalk Programming Language . . . . .	19
2.3	Technologies Used for SQUIMERA . . . . .	22
2.4	Exception Handling and Debugging . . . . .	24
2.5	Summary . . . . .	25
<b>3</b>	<b>Approach</b>	<b>26</b>
3.1	Reuse of Existing Software Development Tools . . . . .	26
3.2	Reuse of Existing Software Libraries and Frameworks . . . . .	32
3.3	Summary . . . . .	33
<b>4</b>	<b>Implementation</b>	<b>34</b>
4.1	Building a Multi-language Virtual Machine . . . . .	34
4.2	Adapting Smalltalk Tools for Other Languages . . . . .	48
4.3	Summary . . . . .	57
<b>5</b>	<b>Evaluation</b>	<b>58</b>
5.1	Live Object Exploration . . . . .	58
5.2	Debugging User Experience . . . . .	61
5.3	Reusing Software Libraries . . . . .	66
5.4	Limitations of SQUIMERA . . . . .	68
5.5	Further Limitations of Our Approach . . . . .	70
5.6	Summary . . . . .	73
<b>6</b>	<b>Related Work</b>	<b>74</b>
6.1	Tools and Integrated Development Environments . . . . .	74
6.2	Cross-language Integration Techniques . . . . .	79
6.3	Summary . . . . .	83
<b>7</b>	<b>Conclusion and Future Work</b>	<b>84</b>

# List of Figures

2.1	Main components of a virtual execution environment . . . . .	15
2.2	Screenshot of a Squeak 1.13u environment from 1996 . . . . .	20
3.1	Architectural comparison of two different IDE approaches . . . . .	27
3.2	Example architecture of our approach . . . . .	31
3.3	Comparing FFIS to our approach with regard to software reuse . . .	32
4.1	Exception handling uses the termination model in Python . . . . .	38
4.2	Example of how PyPy stack frames are restarted in SQUIMERA . . .	44
4.3	Architecture of SQUIMERA's virtual machine . . . . .	46
4.4	Architecture of SQUIMERA's development environment . . . . .	49
4.5	Languages can be switched in the workspace's menu . . . . .	53
4.6	MethodContext stacking in SQUIMERA . . . . .	55
5.1	Inspecting an application-specific object in Squeak/Smalltalk . . . .	59
5.2	Inspecting an application-specific Python object in SQUIMERA . . . .	59
5.3	The inspected object determines syntax and syntax highlighting . . .	60
5.4	Debugging a KeyNotFoundError in Smalltalk . . . . .	61
5.5	Debugging a ZeroDivisionError in Python . . . . .	62
5.6	Unhandled exceptions are returned as result if propagated . . . . .	63
5.7	Interrupting a running Ruby process . . . . .	63
5.8	Debugging a Ruby process while it is running . . . . .	64
5.9	Observing language processes in Squeak's process browser . . . . .	65
5.10	A search tool for Wikipedia which reuses a Python library . . . . .	66

# List of Tables

2.1	Programming languages combined in SQUIMERA . . . . .	22
4.1	General set of primitives of a ForeignLanguagePlugin . . . . .	47

# Listings

2.1	Example implementation of a stack frame in RPython . . . . .	17
3.1	Method override pattern used to retain a tool's original behavior . .	30
4.1	Minimal <code>rstacklet</code> example in RPython . . . . .	36
4.2	Output of the <code>rstacklet</code> example in Listing 4.1 . . . . .	37
4.3	Stack frame walking in the PyPy interpreter for SQUIMERA . . . . .	39
4.4	A simple exception handling example in Python . . . . .	41
4.5	Disassembling Listing 4.4 with Python's disassembler <code>dis</code> . . . . .	41
4.6	An example of other exception handling techniques in Python . . .	43
4.7	Example of an exception being masked by a Python builtin . . . . .	43
4.8	A Python function to check if code is a Python expression . . . . .	51
4.9	<code>printOn</code> : implementation for Python objects . . . . .	52
4.10	<code>printOn</code> : implementation for Ruby objects . . . . .	52
4.11	How SQUIMERA reuses PYGMENTS for syntax highlighting . . . . .	56
5.1	Example method calling out to a Python program via <code>OSProcess</code> . .	67
5.2	Alternative method directly using the <code>WIKIPEDIA</code> library . . . . .	67
5.3	A simple <code>FLASK</code> server example . . . . .	69
5.4	A Smalltalk <i>doIt</i> to start the <code>FLASK</code> example in Listing 5.3 . . . . .	69

# List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CLR	Common Language Runtime
CIL	Common Intermediate Language
DLTK	Dynamic Languages Toolkit
DSL	Domain-specific Language
FFI	Foreign Function Interface
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JIT	Just-in-time
JVM	Java Virtual Machine
JVMTI	Java Virtual Machine Tools Interface
OOP	Object-oriented Programming
REPL	Read–eval–print Loop
RPC	Remote Procedure Call
SDL	Simple DirectMedia Layer
VM	Virtual Machine

# 1 Introduction

Computers have changed the way we think, work, and live. Today, software is used for many different purposes including communication, automation, entertainment, and many more. However, software applications still need to be designed and built by software developers. These software developers are today more expensive than the computers their code runs on. Additionally, software systems are becoming more comprehensive as they are solving more and more complex problems. It thus makes sense to design programming languages and tools to maximize programming productivity.

There are different kinds of programming languages ranging from different abstraction levels starting from C, which is comparatively close to the hardware, to very high-level languages such as Domain-specific Languages (DSLs), which can span multiple abstraction levels and can often only be used for very specific use cases. Usually, the higher the abstraction level of a language, the more it can increase the productivity of developers, as they, for example, do not have to care as much about the underlying operating system.

Dynamic programming languages not only abstract from the operating system. They also enable programming paradigms such as Object-oriented Programming (OOP) and often come with a dynamic type system. Although this dynamicity comes at performance costs in some cases, many developers prefer to use high-level dynamic languages, for example Java, Python, Ruby, and JavaScript, and are willing to trade performance – even when it matters – for platform independence and a better programming experience.

Furthermore, a lot of work has already been put into minimizing the performance differences between lower-level and higher-level languages. Different mechanisms and techniques have also been developed that attempt to combine the best of both worlds to some extent.

Nonetheless, the fact that there are many different programming languages with different features has additional consequences for developers. In general, they often have to choose a language for their programs and are then limited to the features of that language. More importantly, they also have to learn how to use the tools for each language which can be a significant overhead.

In this report, we propose an approach that aims to further improve the programming experience with regard to working with different high-level dynamic languages. To demonstrate this, we have implemented SQUIMERA, an Integrated Development Environment (IDE) for dynamic languages. SQUIMERA is named after Squeak/Smalltalk, an interactive programming system, and Chimera, which is a hybrid creature from the Greek mythology.

## 1.1 The Need for Software Development Tools

Creating software is a comprehensive activity which requires not only knowledge and practice, but also a suitable set of software development tools. These tools are often provided as part of an IDE and usually support developers throughout the entire software development process. Since most software development practices apply to almost all programming languages, one can find similar tools for each language: code editors can provide useful facilities such as syntax highlighting, code completion, code linting, and refactoring utilities. Interactive debuggers simplify the debugging interaction with an application.

However, it is often necessary to rebuild these tools from scratch for a new programming language or to write extensions for existing IDEs in order to adapt existing tools. Consequently, there are often differences between tools of different languages. Sometimes they just look or feel different, sometimes they have different features or behave differently. Nonetheless, whenever software developers learn a new language, they often also need to learn how to use the tools provided by an IDE for that language. This sometimes is the reason why developers prefer to use a familiar programming language for a project rather than using a new one that needs to be learned first, but which might be a better fit for the specific use case.

This problem gets worse in a business context where multiple developers work in a team, or even multiple development teams work on a bigger project. Then, having to learn a new language while also adapting to a new language workbench is a substantial investment for the company. And this might hinder companies to migrate to newer technology.

The discrepancy between different tool sets for different languages states another problem: some programming languages allow developers to do more than others. In most languages, for instance, the conventional approach to write and maintain program source code is to use offline editing tools. Afterwards, the program can then be executed. Whenever a developer wants to change the application, however, it is necessary to restart or reload the program. By contrast, other programming languages, such as Smalltalk, provide incremental compilation and access to language internals which encourages developers to make changes while a program is running [65, p. 22].

## 1.2 The Need for Software Libraries and Frameworks

It is good practice to design software in a modular way, because it supports extensibility as well as reusability in software system [73, pp. 39–64]. This has led to the development of software libraries and frameworks in different programming languages which can be reused for different purposes.

Sometimes for example, a language's standard library does not provide support for a specific file format such as JavaScript Object Notation (JSON). Then developers often implement a JSON parser library for that language. And to build, for example,

a server application, developers often can choose from a list of different server frameworks per programming language.

However, just like tools are different for each language, frameworks and libraries for similar purposes are different. They often provide different Application Programming Interfaces (APIs), differ in features, or behave differently. Additionally, these modular software artifacts also need to be implemented from scratch for each language.

To work around this problem, many programming languages provide Foreign Function Interfaces (FFIs) which allow an application to call routines from a program written in another language. However, these interfaces are usually limited in functionality and only support to call out to the operating system, to C, or other lower-level languages. For this reason, one can often find a lot of wrapper libraries that wrap around foreign function calls and provide a more convenient API in the host language. Nonetheless, it often is inconvenient or sometimes even impossible to test or debug the parts of an application that use FFI.

Another approach to cope with the reusability problem is the microservice architecture which is often applied in much larger software systems that run in the cloud [83]. Instead of writing one complex application, the application is divided into many smaller applications with fewer responsibilities, so-called microservices, which can all be written in different programming languages. The communication between these services usually happens over network, for example via Remote Procedure Calls (RPCs). Even though this approach allows better reuse of software artifacts and supports software modularity, it also comes with certain disadvantages: the network communication introduces a performance overhead as well as other connectivity-related problems, it sometimes is hard to distribute responsibilities in a reasonable way, and also software testing and deployment become more complicated.

### 1.3 Contributions

This work aims to solve two problems: on the one hand, we propose an approach that allows reuse of existing tools for different programming languages, rather than having to reimplement them from scratch. We claim that this makes it easier for software developers to work with more languages, as they do not have to learn how to use the different tools in each case. Instead, they get a consistent programming experience across all languages. On the other hand, we demonstrate how this approach also allows reuse of software components written in different languages in a more convenient way. This gives developers a broader choice when they have to choose which frameworks or libraries they want to use in their applications, as they can now pick components written in different languages. Our key contributions of this work are as follows:

- An architecture to compose multiple languages within the same live programming environment with reflective capabilities for full execution control from within the runtime.
- An implementation of said architecture using PyPy [92] for Python, Topaz [40] for Ruby, and RSqueak/VM [14, 41] for Squeak/Smalltalk.
- Adaptations of the Smalltalk debugger and other tools that work consistently across Smalltalk, Python, and Ruby.
- A more convenient alternative to FFIS that allows developers to reuse software libraries from different languages, without breaking their debugging tools.

## 1.4 Outline

In chapter 2, we provide background information and context for the different aspects of our work. Then in chapter 3, we propose and explain our approach which aims to solve different software development-related problems that were discussed in this chapter. After that, we demonstrate how we have applied this approach as part of our prototype system SQUIMERA in chapter 4. In chapter 5, we give different examples of how the system can be used and evaluate our approach based on these examples. Afterwards, we compare SQUIMERA to related systems and discuss other related work in chapter 6. Finally, in chapter 7, we give a short summary of our approach as well as an overview of future work.

## 2 Context

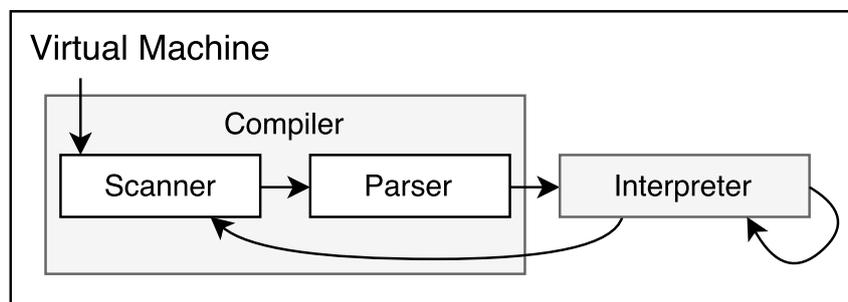
In the following sections, we introduce and explain different technologies and terminologies which later help to understand our approach in chapter 3 as well as the implementation of SQUIMERA as described in chapter 4.

### 2.1 Virtual Execution Environments

A virtual execution environment or *process* Virtual Machine (VM) is a computer program which is able to execute a programming language [98, pp. 13–17]. In the following, whenever we use the term VMs we will refer to *process* VMs. In contrast, *system* VMs emulate hardware and allow the execution of entire operating systems [98, pp. 17–22].

A VM often comes with support for multiple operating systems. This enables platform-independent execution of programs written in the supported language, as virtual execution environment normally abstract from the underlying operating system and the platform architecture [98, pp. 15–17].

**The Components of a Virtual Machine** As illustrated in Figure 2.1, virtual machines usually consist of two key components: a *compiler* and an *interpreter* [98, pp. 85–87]. In general, the compiler translates program code to a representation the interpreter is able to then execute. To be more specific, the execution of a program starts in a part of the compiler which is called *scanner*. It performs lexical analysis on the program code and generates tokens, which is why it is sometimes also referred to as *lexer* or *tokenizer*. In a second step, a so-called *parser* takes these tokens and translates them into an intermediate representation, usually an Abstract Syntax Tree (AST), or a parse tree and then bytecode. In addition to this, virtual



**Figure 2.1:** Overview of the main components of a virtual execution environment

machines may also consist of many other components such as a garbage collector or various optimizers.

An AST is a tree representation of a program. Each node of the tree usually represents an operation, except for leaf nodes which normally represent values. Bytecode, on the other hand, is a more efficient and often optimized representation of a program which does not contain structured information on the program it represents anymore. Instead, it is a direct mapping of instructions in the language to atomic bytecode functions, also referred to as *opcodes*, in the virtual machine. Both, ASTs and bytecode, are typically platform-independent. Instead, they are usually tied to a specific interpreter, which sometimes means that an older or newer version of the same interpreter cannot execute the same bytecode or AST.

Depending on the type of intermediate representation used in a VM, an interpreter either evaluates a program by following the tree structure of an AST, or by using an interpreter loop for bytecodes. AST interpreters have to traverse the tree and jump between nodes which is an overhead compared to sequentially reading in bytecode. On the other hand, it usually is easier to perform optimizations on ASTs than on bytecode. In general, both kinds of virtual machines perform roughly the same. The overall performance depends more on the programming language to execute and the optimizations that can be applied to it in all parts of a VM.

**Executing Object-oriented Programming Languages** In OOP languages, messages can be sent to objects [28, pp. 121–123]. As an example, sending *address* to a *Person* object may return the person’s address. To facilitate such a *message send*, a virtual machine often uses a *Frame* data structure [78, p. 114]. Such a *call frame* usually stores the receiver of the message as well as a reference to the currently executed method. Message sends, however, may call other messages as well. In our example, the *address* message may assemble the person’s address by combining the person’s name, street, zip code, and country. For this reason, a frame also stores the sender frame and is executed as a child of another frame. This way for example, the virtual machine knows for which frame the result of a person’s street is relevant. Therefore, frames are stacked according to the program execution [96], which is also known as *call stack*, *program stack*, or *execution stack*.

Listing 2.1 shows an example implementation of a frame in RPython for a bytecode-based language implementation. On initialization, the *bytecode*, the *receiver*, and a list of *child\_frame* are provided. In this frame implementation, the *receiver* is stored as the first value on the *value\_stack*. Further, the *Frame* class implements an *interpret* method which evaluates the frame and returns the result. This method contains the interpreter loop for which it first initializes a temporary variable *pc* for the program counter. The interpreter loop is implemented as a *while-true* loop. First, the *instruction* is retrieved from *bytecode* using the program counter. In a next step, this counter is incremented, so that it points to the next position in the *bytecode*. After that, the corresponding bytecode implementation for the *instruction* is looked up.

The *CONST\_INT* instruction, for example, takes the next instruction as an integer value, increases the counter one more time, and wraps and pushes the value onto

**Listing 2.1:** Example implementation of a stack frame in RPython

---

```

class Frame:

    def __init__(self, bytecode, receiver, child_frames):
        self.bytecode = bytecode
        self.value_stack = [receiver]
        self.child_frames = child_frames or []

    def interpret(self):
        pc = 0
        while True:
            instruction = ord(self.bytecode[pc])
            pc += 1
            if instruction == CONST_INT:
                value = ord(self.bytecode[pc])
                pc += 1
                self.push(W_IntObject(value))
            elif instruction == ADD:
                self.push(self.pop().add(self.pop()))
            elif instruction == POP:
                self.pop()
            elif instruction == DUP:
                self.push(self.peek(1))
            elif instruction == JUMP_IF:
                if self.pop().is_true():
                    pc = ord(self.bytecode[pc])
                else:
                    pc += 1
            elif instruction == INTERP_FRAME:
                index = ord(self.bytecode[pc])
                result = self.child_frames[index].interpret()
                self.push(result)
            elif instruction == EXIT:
                break
            # more bytecodes ...
            else:
                raise RuntimeError('Unknown bytecode: %s' % instruction)
        if len(self.value_stack) == 1:
            return self.pop()

# ...

```

---

the `value_stack`. The `ADD` instruction pops the top two values from the stack, adds them together, and pushes the result again. There usually are various opcodes to further interact with a value stack, such as a `POP` bytecode to pop or a `DUP` bytecode to duplicate the top value. A `JUMP_IF` bytecode, on the other hand, manipulates the program counter if the top value is `true`, otherwise it just increments the counter by one. Therefore, this bytecode can be used for conditional forward, but also backward jumps in the bytecode. In this example, a frame can have children which can be executed with the `INTERP_FRAME` bytecode. This instruction looks up the index in the `child_frame` list, starts the interpretation of the frame, and pushes the result back onto its own `value_stack`. Moreover, there usually are bytecodes that can escape the interpreter loop, such as `EXIT` in this example. In addition to that, there are many more instructions, for example, language-specific instructions that add support for more data types and operations.

**Performance Optimizations** To increase the performance of a programming language, various optimization techniques can be applied to almost all components of virtual machines. The compiler might be able to produce an optimized AST or bytecode by, for example, removing redundant instructions.

More importantly, “interpreting bytecodes is slow” [29]. Consequently, a lot of work has been put into optimization techniques for interpreters. Polymorphic inline caches can be used to significantly speed up method lookups for example [57]. Just-in-time (JIT) compilation is another optimization technique.

A JIT compiler usually first tries to identify reoccurring control flow in the runtime system by gathering profiling information. Then, it compiles the corresponding code to machine code which is then cached and used instead of the original code during the program execution. This machine code often combines multiple interpretation steps into one, which further increases performance. Moreover, the compiler can apply additional optimizations including method inlining, dead code elimination, or heap allocation elimination. However, a JIT compiler has to insert checks, so-called guards, to ensure that the machine code behaves exactly as the interpreter would behave. Whenever such a guard fails, the execution environment switches back to the interpreter.

According to Aycock [3], McCarthy’s Lisp [71] from the 1960s was the first system to support JIT compilation. Around twenty years later, new JIT optimization techniques were developed to efficiently execute the dynamic languages Smalltalk and Self [3, 33]. Today, sophisticated JIT compilers can be found in many virtual machines, including the Java Virtual Machine (JVM) [29], LuaJIT [82], and PyPy [13].

Furthermore, AST interpreters can optimize the program execution by, for example, rewriting the abstract syntax tree with specialized nodes that are faster to execute. Würthinger et al. presented an approach for self-optimizing AST interpreters [118] which was further implemented as part of the Truffle language implementation framework [117].

## 2.2 The Smalltalk Programming Language

Smalltalk is a dynamic and object-oriented programming language. It was designed and created in the Xerox PARC Learning Research Group by Alan Kay, Dan Ingalls, Adele Goldberg, and others during the 1970s [49]. The first generally released language specification is called Smalltalk-80. Unlike many other programming languages, Smalltalk-80 includes a full-fledge programming environment with different software development tools. In 1997, Kent Beck published a book on “Smalltalk Best Practice Patterns” [10] based on these tools which has highly influenced the way developers think about and design software in general. Moreover, the original set of Smalltalk tools can still be found in many other programming environments. All of this indicates that these tools are mature and still state-of-the-art, even though Smalltalk was designed almost 40 years ago.

Since then, many Smalltalk-80 implementations have evolved including Squeak/Smalltalk [59], VisualWorks [58], GemStone/S [23], GNU Smalltalk [43], and many others. Some of these implementations are for educational or research purposes, others are used in a purely commercial context. IBM’s VisualAge is another example for a Smalltalk-based and commercially used IDE which later supported other languages, such as Java [25], and eventually led to the development of the Eclipse framework [32] which today is still in active development and heavily used by many Java developers.

**The Smalltalk Virtual Machine** A Smalltalk system traditionally consists of two components: an image file and a virtual machine. The image file contains a memory snapshot which the VM loads when the file is opened. In this memory snapshot, the last state including all objects of the environment is persisted and whenever the user saves the environment, the VM creates a new memory snapshot and saves the result to disk.

There are different Smalltalk virtual machine implementations for different Smalltalk implementations. The current default for many Smalltalk flavors and derived languages was recently renamed from Cog VM [76] to OpenSmalltalkVM [77], is based on the original Squeak/Smalltalk Virtual Machine [59], and comes with a JIT compiler. This virtual machine is mostly written in Slang, a subset of the Smalltalk language. For this subset, there is a compiler, which is also written in Smalltalk and which is able to translate Slang code to C code. This C code can then be compiled to machine code for different operating systems and architectures.

The VM uses a bytecode interpreter loop which fetches and executes the next bytecode of a `CompiledMethod` object. For this, the interpreter also has a reference to the currently active `MethodContext` object, which represents a message send in Smalltalk and in which it can look up the next instruction. Therefore, when an image is opened, the virtual machine only has to identify the last active context and then starts to execute the next bytecode of the corresponding `CompiledMethod`.

In addition to the bytecode set, the VM also implements primitive methods. These primitive methods can be invoked from within the Smalltalk environment and are used to facilitate the communication between the programming environment and

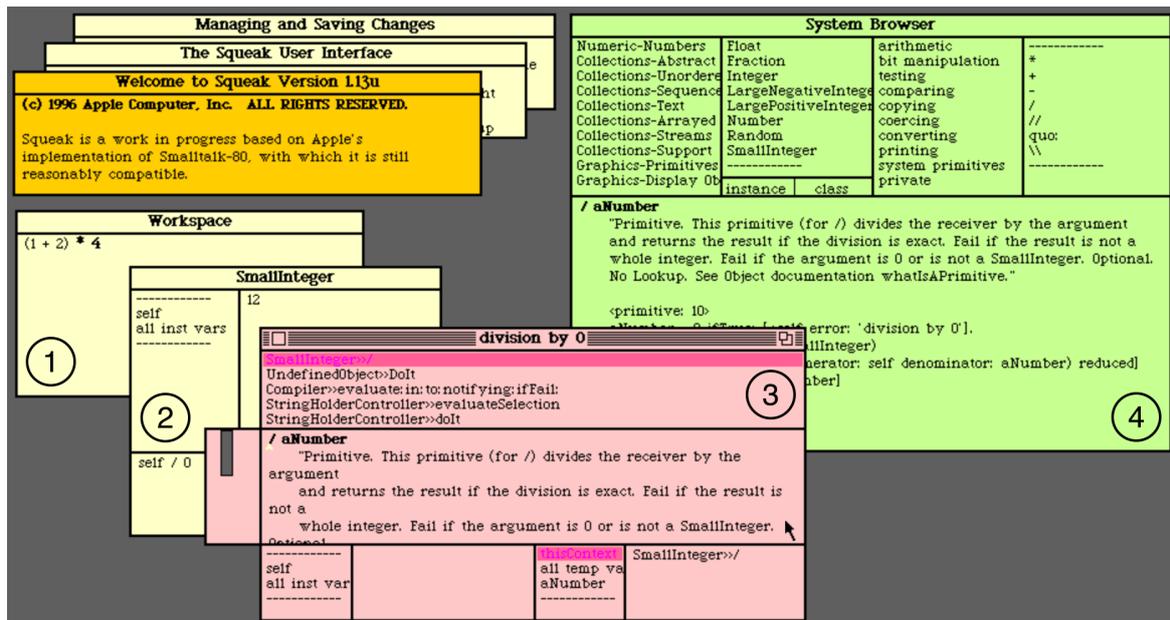


Figure 2.2: Screenshot of a Squeak 1.13u environment from 1996

the virtual machine. The Smalltalk-80 specification defines a bytecode set as well as a list of primitives that a virtual machine needs to implement to be able to open a minimal Smalltalk image. Moreover, the list of primitives can be extended by VM-level plugins. This way it is possible to add support for new features that the image can then access.

**The Smalltalk Environment** Figure 2.2 shows a screenshot of a Squeak 1.13u environment which was released more than 20 years ago. The different windows opened on the display screen were originally referred to as *views* [49]. Most of them are examples of the different Smalltalk tools, except the three overlapping windows on the top left which contain Squeak/Smalltalk’s license and useful information on how to use the environment.

The most basic tool is called workspace ①. Its main purpose is to provide documentation or templates of common expressions [48, pp. 111–112]. It can only hold text, which can then be evaluated by the developer using different mechanisms such as a *doIt*, a *printIt*, or an *inspectIt*. A *doIt* simply executes code, while a *printIt* also displays the result of the execution in the text field [48, pp. 106–111]. An *inspectIt*, on the other hand, opens an inspector window on the result. All mechanism work in the same way: if code is selected in the text field, they execute the selection. Otherwise, they execute the code on the line of the cursor. In addition to that, a modern Squeak/Smalltalk environment also supports more mechanisms such as *exploreIts* which open the object explorer tool instead of the inspector, or *debugIts* which open the debugger.

The inspector tool ② is designed for object inspection purposes and provides a live view of object internals [48, pp. 144–154]. For this reason, it contains a list of

all instance variables which can be inspected further. Moreover, the inspector tool comes with a text field on the bottom that behaves similar to a workspace, but has the `self` keyword bound to the inspected object. The object explorer, which was introduced in a later version of Squeak/Smalltalk [102], is conceptually similar, but presents the information in a tree structure [79, pp. 134–135].

Window ③ shows the interactive Smalltalk debugger [48, pp. 388–399]. It was opened after a *dolt* was performed in the inspector, which in turn attempted to divide the `SmallInteger` object 12 by 0. This caused a division by zero error which is indicated in the window title of the debugger. The top part of the view lists the different stack frames that led to the error. In the middle, a text field shows the code that is being executed in the selected frame. Changing the frame selection using the list on the top, also changes the source code shown in the text field. On the bottom left of window ③, there is an inspector-like component that is attached to the receiver of the message, which is the `SmallInteger` object 12 again. The bottom right half contains another variant of the inspector which shows further information on the execution context. This includes *thisContext* referencing the original `MethodContext` object which represents a frame in Smalltalk, as well as a list of all temporary variables. In this case, there only is the temporary variable `aNumber` which holds a reference to the `SmallInteger` object 0 that was used as the divisor. Furthermore, the debugger allows the user to proceed with the execution, to restart a frame, or to step to the next line of code via a context menu. This functionality is especially useful, because the code shown in the text field is modifiable. This means that developers are able to modify a Smalltalk application with the debugger tool while it is running, also known as *edit-and-continue* debugging which we will explain in section 2.4 in more detail.

The window ④ on the right shows the Smalltalk system browser [48, pp. 218–291]. The upper half consists of four lists. The first list contains all class categories, the second the classes for the selected category. Then, the different method protocols are listed for the selected class, and finally there is a list with all methods for the selected method protocol. A text field shows the code for the selected method on the bottom of the system browser window. With this tool, it is possible to create new classes in specific categories and to add methods in specific protocols to them. This can be used to develop comprehensive software applications in a structured way. Different context menus add useful features which further support the software development process. With these features, developers are, for example, able to see the class hierarchy of a given class, to list the senders or other implementors of a method, and to refactor different parts of their applications conveniently.

In addition to the tools shown in Figure 2.2, Squeak 1.13u also shipped with a transcript [48, p. 65], which is a tool similar to the workspace but with the main purpose of logging messages, a file list browser [48, pp. 439–448] which lists directories and files stored on disk, and a change list tool [48, pp. 472–477] to browse the changes file which Squeak/Smalltalk maintains to keep track of all changes in the system.

At the time of writing, the latest stable release of Squeak/Smalltalk is version 5.1. It contains improved versions of the tools presented before. Besides, more

**Table 2.1:** Programming languages combined in SQUIMERA

Language	Creator	Released in	Default VM	RPython alternative
Squeak/Smalltalk	A. Kay et al.	1996	OpenSmalltalkVM	RSqueak/VM
Python	G. van Rossum	1991	CPython	PyPy
Ruby	Y. Matsumoto	1995	Ruby MRI	Topaz

tools have been added that support the development process even further [102]. A process browser tool allows users to inspect, debug, suspend, and terminate Smalltalk-level processes. For a convenient interaction with Smalltalk TestCases, Squeak/Smalltalk’s test runner can be used. With the Monticello browser, it is possible to manage Smalltalk code in local and remote repositories.

## 2.3 Technologies Used for Squimera

Table 2.1 gives a quick overview of the different programming languages and some of their implementations we intend to use for SQUIMERA. For all languages, there is an alternative implementation in the language implementation framework RPython [1]. We use Squeak/Smalltalk as our Smalltalk environment, because there is an RPython-based VM called RSqueak/VM [14, 41]. We intend to pair this system with the following two other language implementations: The most prominent virtual machine implemented in RPython is PyPy [92], which is an implementation of the Python programming language. The second language implementation in RPython is Topaz [40], which is a VM for Ruby.

### 2.3.1 Squeak/Smalltalk and RSqueak/VM

Squeak/Smalltalk [59] is a Smalltalk dialect directly derived from the Smalltalk-80 language specification. It is originally developed by Alan Kay, Dan Ingalls, and others, most of which were also part of the Xerox PARC Learning Research Group which designed Smalltalk-80. The default framework for Squeak’s graphical user interface is Morphic [70], which originally was developed for the Self programming language. Over the years, many projects have evolved from Squeak/Smalltalk including Newspeak [19], Etoys [44, 62], Scratch [88], Pharo [12], and Babelsberg/S [39].

As mentioned before, RSqueak/VM is a VM for Squeak/Smalltalk written in RPython. It is developed and maintained by the Software Architecture Group at the Hasso Plattner Institute in Potsdam. As part of various seminars, students have implemented different concepts, ideas, and performance optimizations in RSqueak/VM [41].

### 2.3.2 Python, PyPy, and RPython

Python is a dynamic, object-oriented, and high-level programming language created by Guido van Rossum [111]. It has a dynamic type system, an automatic memory management, and comes with a comprehensive standard library which makes it especially useful as a scripting language [7]. Its default virtual machine is called *CPython*. According to the “TIOBE Programming Community Index for April 2017” [107], it is on position five of the most popular programming languages.

PyPy [92] is an alternative implementation of the Python language. Because of its JIT compiler, it often executes Python programs much faster than CPython. However, PyPy is not completely feature-identical and therefore does not support all Python packages. Nonetheless, the PyPy maintainers constantly add support for more packages while ensuring that compatibility for already supported packages does not break. PyPy’s JIT compiler is a tracing JIT which traces the execution of a program to detect frequently executed instructions which are then compiled to machine code.

The RPython framework is part of the PyPy project and is a tool chain for building implementations of dynamic programming languages. Technically, RPython is a subset of Python which can be compiled to C and then to machine code. For this to work, RPython enforces a number of constraints on the way the Python language can be used, for instance with regard to different data types. As part of a compilation process, RPython determines low-level C types for each Python data type which can fail when a developer has mixed different types, for example, in a Python `list`. Additionally, the framework contains the JIT compiler used for PyPy which can be reused in other language implementations. Therefore, RSqueak/VM can also leverage this JIT compiler to increase run-time performance.

### 2.3.3 Ruby and Topaz

Ruby is a dynamic, object-oriented, reflective, and high-level programming language created by Yukihiro Matsumoto [42]. Similar to Python, it has built-in memory management, supports dynamic types, and includes a powerful standard library. Similar to Smalltalk, on the other hand, everything in Ruby is an object, because Matsumoto wanted Ruby to be “more object-oriented than Python” [101]. The default virtual machine for Ruby is called *Ruby MRI*, which stands for “Matz’s Ruby Interpreter” and which sometimes is also referred to as *CRuby*. It is on position ten of the most popular programming languages in the “TIOBE Programming Community Index for April 2017” [107].

Topaz is an alternative Ruby implementation. Since it is written in RPython, it features a tracing JIT compiler which allows it to execute Ruby programs faster than CRuby. Topaz only supports a limited set of Ruby programs, as its implementation is still missing many Ruby MRI features.

## 2.4 Exception Handling and Debugging

Mistakes made by software developers can lead to misbehavior within applications, more commonly known as software *bugs*. These bugs eventually cause an *infection* in the program, which in turn can then cause a program *failure* [120, p. 21]. On the other hand, sometimes parts of an application are allowed to fail. For this, a program can throw *exceptions* for which developers can then, for example, implement fallback behavior.

**Handling Exceptions in an Application** In order to be able to deal with exceptions in a program, programming languages provide a mechanism called *exception handling*. This mechanism has to be considered in many important parts of the language, such as control flow, coroutines, concurrency, and polymorphism [22]. With exception handling, it is, for example, possible to implement fallback behavior whenever a certain error or exception occurs. Exceptions thrown in an application that do not have a fallback, are often called *unhandled exceptions*.

According to Yemini and Berry [119], there are four different models for exception handling: non-local transfer, termination, retry, and resumption. When an error is detected, the control flow can be redirected dynamically to any location in a program by using, for example, a *goto*-like statement to jump to a variable position. This is referred to as a non-local transfer. However, experiences gained by the use of these kind of transfers in the PL/I programming language show that non-local transfers are error-prone and make it harder to maintain programs [68, p. 102]. In the termination model, which is the most popular according to Buhr et al. [22], the program is restricted to continue in an exception handler in the current stack frame or one of its parent frames instead. The restart model allows restart of a failed operation. However, this behavior can easily be recreated using the termination model and loops [45, p. 834]. Finally, the resumption model provides the ability to transfer control flow to an exception handler and then back to the point where the exception was raised.

Although exception handling is somewhat related to software bugs, it does not avoid or fix them. Instead, developers have to *debug* their programs.

**Finding and Fixing Software Defects** There are, of course, different techniques to identify software problems and different ways to fix them. Zeller [120, pp. 200–225] describes two approaches which help to observe misbehavior in programs. One way is to introduce logging functions in a program, which print useful information about the internal state of an application to the console or into a log file. However, these functions often clutter program code and sometimes, it is hard to follow the state when logging is too verbose. Another option is to use a *debugger*, which can usually be attached to the process executing the program and which can provide different facilities to inspect, but also to fix problems. The former technique is also known as *printf debugging*, the latter as *interactive debugging*.

Debugging a program after it has crashed is often referred to as *post-mortem debugging* [47, 81]. This type of debugging often requires that a bug is easy to

reproduce, because developers might not be able to debug the program on the system it failed on. Instead, they regularly use a development machine which might not be set up in the same way the original system is set up, which can make it more difficult to reproduce a problem.

While logging functions only allow post-mortem debugging, some interactive debuggers support more powerful debugging techniques such as *edit-and-continue debugging* [36] which is also sometimes referred to as *fix-and-continue debugging* [120, pp. 223–224]. As the name suggests, this kind of debugging allows users to inspect, modify, and resume a program while it is running. In Smalltalk, for example, the interactive debugger is opened by the environment when an unhandled exception is detected. Then, developers are able to see the code that triggered the exception, as well as the current execution context including all stack frames. With that, they are able to fix the bug in their application and finally restart one of the stack frames that led to the exception. When the changes proposed by the developer fixed the bug, the application continues to run. If not, another debugger window is opened. However, this debugger feature highly depends on the capabilities of the underlying execution environment, because it usually requires the ability to change classes and methods at run-time, also known as *code hot-swapping* [35].

## 2.5 Summary

Bytecode loops are a common and well-understood technique to implement efficient high-level dynamic languages, such as the Smalltalk programming language upon which our work is built. Smalltalk was chosen for its mature set of tools, which include a workspace, object inspection tools, and an interactive debugger. RSqueak/VM is a VM for Smalltalk and written in the RPython framework, which has been used for implementing many other high-level languages including Python and Ruby. Important aspects of our work on combining multiple languages are exception handling and the ability to change code at run-time during debugging.

## 3 Approach

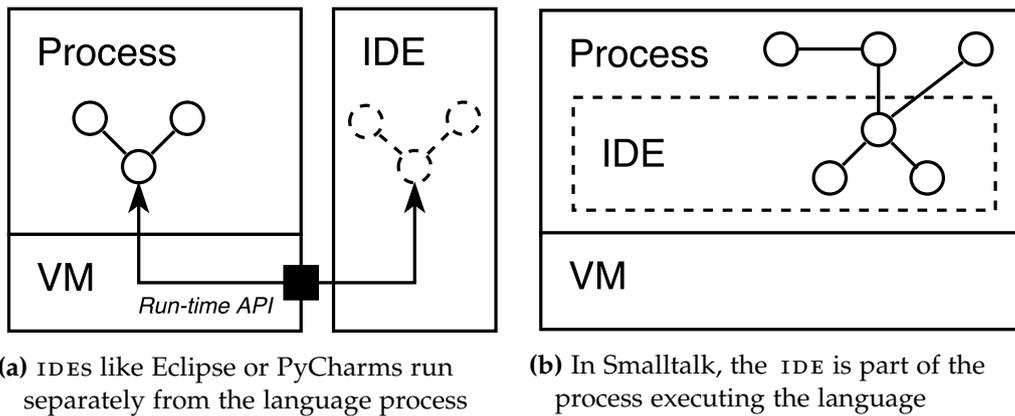
In chapter 1, we described the needs of software developers with regard to programming languages, and highlighted pain points they encounter when starting to work with a new language. The tools for each programming language are different, but developers rely on them to build software efficiently. The same holds true for software libraries and frameworks for various software development tasks, which developers build to promote modularity and allow code reuse. As a result of that, frameworks and libraries that have proven to be useful in one language are often ported or recreated in other languages. This, however, means that all components, including tools and reusable software artifacts, usually are written from scratch in each language, which is also one reason why they often differ in functionality.

We propose an approach which aims to provide solutions for these problems and which consequently attempts to result in a better programming experience for developers. Because there are many different types of programming languages that follow various programming paradigms, we focus on high-level dynamic programming languages in our approach.

### 3.1 Reuse of Existing Software Development Tools

Instead of having to build new tools, we suggest to reuse existing tools on language level. For one, this relativizes the problem for developers of having to adapt to new tools whenever they switch to another programming language. Ideally, developers are already familiar with the tools and only need to learn the syntax and concepts of another language. Additionally, this eliminates the need to write tools from scratch. Moreover, in a perfect scenario, a developer improves one tool and all other developers can benefit from that improvement, no matter which programming language they use. This perfect scenario might come with new challenges or might be hard to reach, but reusing tools that already exist is one step towards this goal.

**Tools Are Usually Not Part of the Language** In most programming languages, the tools are not part of the language implementation, which is one reason why they can have different functionalities. Some languages ship tools as part of their standard library, but as long as these tools are not deeply integrated with the execution environment, they are limited in use. For some language implementers, tools or certain features might not even be important enough, so they leave the tool building activity to, for example, language users or third parties. Consequently, the common approach of building IDEs, as shown in Figure 3.1a, is to have them separate from the process that executes the language. In this setup, an IDE commu-



**Figure 3.1:** Architectural comparison of two different IDE approaches

nicates through some run-time API with the execution environment, which again often provides very different capabilities for inspecting and controlling programs that are being executed. To interact with objects at run-time, IDEs often operate on some kind of proxy objects that represent the real objects living in the language process. Many IDEs allow adaptation of their tools, but not all of their features always match with those provided by the language of interest. And this mismatch often causes a rather inconvenient experience for developers.

Some companies are commercially successful with building mature tools and IDEs for professionals, but they are usually also limited to what the language and its execution environment allow them to do.

In Smalltalk systems, however, the programming environment is part of the language and therefore runs in the same process, as illustrated in Figure 3.1b. Moreover, the language is mostly self-contained and designed in a generic way. Instead of having to call out to a run-time API which might limit them in functionality, Smalltalk tools have direct access to language internals. This allows for very powerful features that support developers in building software more interactively. Because of its generic and adaptable design and its feature-rich tools, we believe a Smalltalk environment makes a good reusable IDE for other programming languages.

Furthermore, there is a renewed push in current research to allow developers to use live run-time data that can be explored and manipulated to understand and extend software systems [18, 112]. A Smalltalk environment also is able to provide such capabilities.

**Composing Language Implementations** In order to be able to reuse tools from a Smalltalk environment, multiple languages need to be combined in one virtual execution environment. Moreover, language implementation frameworks such as Oracle’s Truffle framework [115] or the RPython framework [1] are increasingly used to implement alternative execution environments for programming languages. Therefore, we can build a VM supporting multiple languages with relatively low

effort if we combine a Smalltalk-80 implementation written in such a framework with one or even more other language implementations written in the same framework. Because there is RSqueak/VM, a Smalltalk VM written in RPython, we choose RPython for our approach.

However, this kind of language composition raises the question of how to execute two or more languages at the same time. Even though Smalltalk was designed many years before multi-core processors were introduced, processes are part of the language to support parallel computing [49, pp. 251–257]. As we will demonstrate further in chapter 4, it is possible to coordinate the execution of a non-Smalltalk programming language with the Smalltalk process scheduler, ultimately creating new Smalltalk-level processes for each foreign language invocation. Since the environment’s user interface is also updated by such a Smalltalk-level process, it is possible to interact with the environment while programs written in other programming languages are being executed.

Nonetheless, the Smalltalk scheduler can only coordinate the execution of different Smalltalk-level processes if these processes complete, or, in case of a long-running process, yield at some point. Therefore, we need to ensure that language processes can be suspended and continued at a later point in time. Since they are responsible for running code in a different interpreter loop, this implies that we need to make sure that we can suspend and continue interpreter loops at any time. This mechanism is supported by some programming languages through the notion of continuations [89], but relying on that would rule out many other languages. Instead, it would be better if there is a continuation mechanism which works on interpreter-level.

Most interpreters, however, are implemented as stack machines that manage the execution of code in nested stack frames [96], which is why they are sometimes referred to as *stackful* interpreters [11, p. 390]. But this makes it rather impractical to retrofit continuation capabilities into a language without changing or extending its specification.

However, there is a mechanism we can use which is not specific to a language, but to the underlying language implementation framework. In order to enable stackless features in PyPy, RPython implements *stacklets* [106] as part of its `rpython.rlib.rstacklet` package. Stacklets are C-level coroutines [64, pp. 193–200] that can be used as a one-shot continuation. In PyPy, *continulets* and *greenlets* are built on top of stacklets. If we use them in the same way in our language processes, we can switch between the main thread and coroutines that run bytecode interpreters as we wish.

In addition to this, we also need to find a way to ensure that an interpreter eventually suspends itself and yields back to Smalltalk. Some interpreters already have the capability to periodically perform specific actions which can be used for this purpose. For implementing this, a simple counter is often used as a trigger and executes periodic actions, for example, after a given number of bytecodes. Therefore, if an interpreter does not support periodic tasks, it is straightforward to add this feature when integrating the interpreter.

Furthermore, Smalltalk vms have support for plugins which extend a virtual machine with new primitives. Therefore, it makes sense, not only for modularity reasons, to bundle the same set of primitives needed for a language implementation in a plugin. This will also make it possible to change the programming languages in the Smalltalk environment by simply changing the plugin the environment communicates with.

Composing language implementations, on the other hand, opens up the ability to retrofit features that allow developers to directly control the execution of a program written in a foreign language. In case of interpreted languages, it is now possible to manipulate the interpreter loop in such a way that it is possible to restart a specific frame for example. This ability is later required to enable edit-and-continue debugging as described in section 2.4.

Moreover, programming languages can use different exception handling models. In Smalltalk, exception handling is implemented in the language, which means that the virtual machine does not need any capabilities to handle language-level exceptions. The implementation follows the resumption model of exception handling which is also needed to allow edit-and-continue debugging.

In contrast, Python, for example, uses the termination approach of error handling and does not allow developers to repair the cause of an error nor does it support to retry a failing operation [110, p. 20]. Instead, exceptions are propagated to parent execution frames which is also known as *stack unrolling* or *stack unwinding* [22]. This, however, needs to be avoided to be able to provide the same debugging experience as in Smalltalk. If the debugger is opened after the stack is unrolled, it would not be possible to see the root cause of the problem, because the corresponding frame and intermediate frames have already been removed from the call stack. But since we also do not want to change the exception handling mechanism of the Python language, we have to determine whether an exception is handled or not by one of its parent frames at the time it is raised and before the stack is modified.

In order to ensure that all of these different features exist that are required for proper tooling support and to match the Smalltalk programming experience, we can introduce appropriate abstractions for foreign language implementations. Once the abstract interface is implemented for a given language, the tools will then be ready to use for that language.

**Bridging Between Smalltalk and Other Languages** This interface, however, is not only part of the virtual machine implementation. It also has to reach into the Smalltalk environment. The reason for this is that the Smalltalk tools need to communicate with objects from foreign languages, so some kind of bridge between the languages is required. First, it needs to be possible to perform foreign method or function calls from Smalltalk. For this reason, we propose to map the Smalltalk semantics to foreign languages. We claim that this is possible, because Smalltalk is designed in a generic way and is therefore flexible enough for semantic mappings like this. Then, we need to introduce a new Smalltalk class which is used by the vm to represent objects of a foreign language in the environment. This new Smalltalk class has to inherit from Smalltalk's `Object` class. Only this way we can ensure

that foreign language objects implement the Smalltalk meta-object protocol which the tools use to communicate with all objects. After that, we can start overriding methods that are part of this protocol, so that they return information of the foreign language object they represent. These method overrides, however, need to return Smalltalk objects, so there also needs to be some facility that can be used to convert objects between the languages. In chapter 4 we will demonstrate, that it is sufficient to support the conversion of primitive data types in order to make the tools work.

**Adapting the Smalltalk Tools** After integrating another language like this into the environment, it is possible to adapt the Smalltalk tools. Instead of extending the original tools, we introduce subclasses for each tool and extend it by overriding existing methods and by adding new ones.

In order to not break the tools for Smalltalk objects, we consistently add a check at the beginning of each override method, as shown in Listing 3.1. This check only needs to determine if the tool operates on a foreign language object and if it does not, we call and return the original method from the super class. For convenience reasons, `isForeign` is added as an extension method to the `Object` class in Smalltalk, and is overridden in `ForeignObject` as well as in all tool adaptations.

The simplest tool to adapt is the Smalltalk workspace. Since its main job is to evaluate code, only the method which is responsible to execute user input needs to be implemented. Instead of using the Smalltalk compiler, it has to redirect the code to the corresponding foreign language plugin. The plugin then starts a new Smalltalk-level process which evaluates the code using the language's interpreter and finally returns the result. To make Smalltalk *printIts* work, the corresponding language class in the Smalltalk image only needs to override the `printOn:` method which is used to generate the print string representation in Smalltalk. This is all that needs to be done to adapt the first tool.

Similar to the workspace, the inspector and object explorer tools can be adapted to add support for *inspectIts* and *exploreIts*. They mostly operate directly on the object on which they are opened on, so we just have to make sure that the foreign language class implements the corresponding methods, such as a method to return all instance variables and a method that retrieves a value of an instance variable of an object. Lastly, we again add subclasses for each tool and override the method that is used when evaluation code in the tools code input fields. Similar to the workspace, the code to evaluate needs to be sent to the corresponding foreign language plugin which then executes the code accordingly.

**Listing 3.1:** Method override pattern used to retain a tool's original behavior

---

```
methodName
  self isForeign iffFalse: [ ↑ super methodName ].
  "Code for foreign language objects..."
```

---

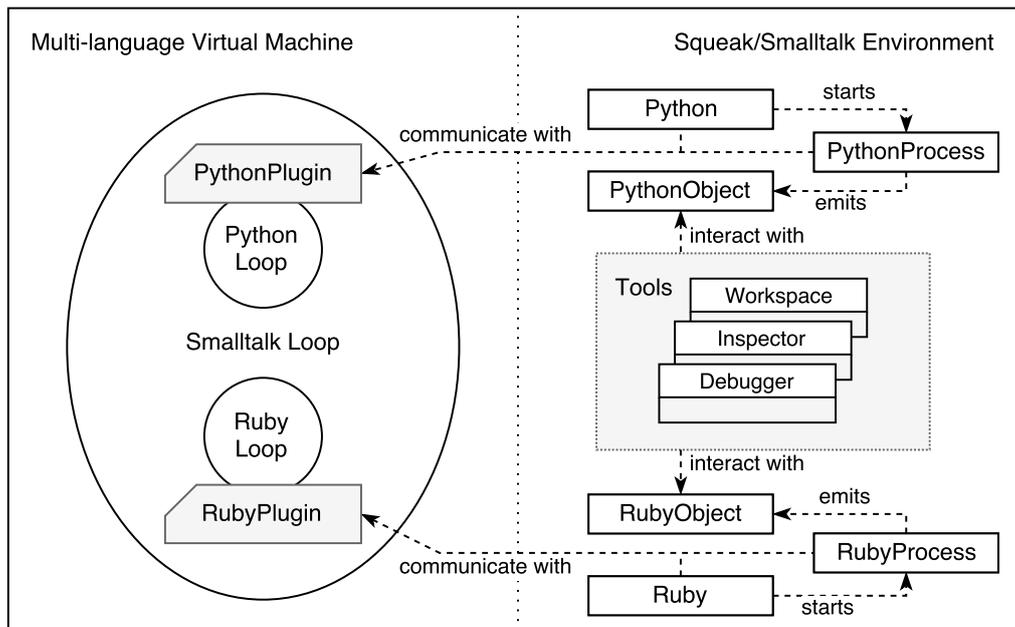


Figure 3.2: Example architecture of our approach

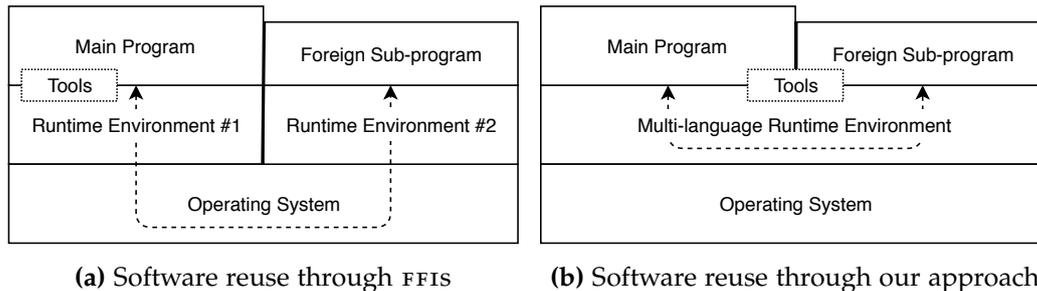
Adapting more tools is straightforward, because it works in a similar fashion: First, create a subclass of the original tool, then override methods appropriately as well as methods of the foreign language class that are used by the tool. For the Smalltalk debugger, for example, only a reasonable amount of methods need to be overridden. Even though the debugger is a lot more complex than the inspection tools and reaches deeper into language internals, this demonstrates that it is indeed straightforward to adapt other Smalltalk tools.

In chapter 4, we will demonstrate how we adapted Smalltalk tools independently from the foreign language, so that subclassing tools and overriding some of their methods as described above does not need to be done for every language that is added to the environment. This does add slightly more complexity to the tools, but decreases the work to provide tools for a new language significantly.

**Architecture** As briefly mentioned before, we propose an architecture as part of our approach. An overview is shown in Figure 3.2 and later used for the implementation of the SQUIMERA system. The system conceptually consists of two main components: a virtual machine and a Squeak/Smalltalk environment. The VM has support not only for Smalltalk, but also for other languages such as Python and Ruby. For the communication between the environment and the runtime, the virtual machine provides a plugin for each non-Smalltalk language. All of these plugins are implemented consistently as they all inherit from an abstract foreign language plugin. In Squeak/Smalltalk, we introduce different classes for each of those foreign languages. The Python and Ruby classes, in this case, facilitate the communication with the VM and can, for example, create a new PythonProcess or RubyProcess respectively. These process objects represent the execution of a language and can

be used by the Smalltalk scheduler to resume them, but also to retrieve information about the current state of the execution. The classes `PyObject` and `RubyObject`, on the other hand, are used to expose foreign objects inside the environment. Additionally, we introduce subclasses from Smalltalk tools and add some other classes that are required for tool support.

### 3.2 Reuse of Existing Software Libraries and Frameworks



**Figure 3.3:** Comparing FFIS to our approach with regard to software reuse

Similar to adopting existing tools, it is valuable to developers to be able to reuse existing software artifacts. In fact, it would be very useful if developers are not limited to choose from a list of frameworks and libraries written in the language they develop in, but also from lists of modules written in other languages. In the worst case, they have to recreate something that already exists. As mentioned in section 1.2, some languages provide FFIS which allow an application to call into other programs, but these interfaces are often inconvenient to use. Figure 3.3a demonstrates why. Usually, an FFI call is somehow propagated to the operating system which then starts a sub-program in another process on top of another execution environment. The tools, however, can only interact with the main program's run-time, which is why they are limited when it comes to debugging FFI calls for example.

**Further Benefits of Our Tool Reuse Approach** On the other hand and as part of adopting the Smalltalk tools described in section 3.1, we already had to bridge between the two languages. Therefore, it is already possible to call out to foreign languages and to convert primitive data types back and forth. But this integration is, of course, not limited to adapting tools. Instead of calling internal functions of the foreign language, we can also call out to libraries that are implemented in this language. Hence, we are immediately able to reuse software libraries of a foreign language in Smalltalk. In section 5.3, we will give a few examples of how this can be used. More importantly, the tools are able to operate on both, the main program

as well as the sub-program, because everything shares the same virtual execution environment, as visualized in Figure 3.3b.

However, additional work is needed to support the reuse of software frameworks. Unlike in libraries, control flow is usually inverted in frameworks [38]. This implies in our case that it also needs to be possible to call from foreign languages back into Smalltalk to be able to use foreign frameworks.

Furthermore, this additional work has another positive effect. When integrating more than one foreign language with Smalltalk, for example Python and Ruby, we are also able to reuse Ruby libraries and frameworks in Python and vice-versa. The Smalltalk language then acts as a communication layer between the two languages while providing development tools for both at the same time.

### **3.3 Summary**

Our approach is based on interpreter composition which allows us to build a multi-language virtual machine for a Smalltalk programming environment. With this, we can adapt various Smalltalk tools for developing other languages and provide a consistent programming experience. As another result of this integration, our approach also allows the reuse of software libraries and frameworks written in different languages in a convenient way and with debugging support across all integrated languages.

## 4 Implementation

In this chapter, we explain how we have implemented our SQUIMERA prototype system based on the approach described in chapter 3 and highlight important implementation details. For the different programming languages, we are reusing existing implementations written in the language implementation framework RPython [1].

SQUIMERA's multi-language VM is based on RSqueak/VM and needs less than 1,600 SLOC of additional RPython code. Of these, roughly 500 SLOC are needed for each specific language integration, while more than 500 SLOC are shared between all integrated languages. Furthermore, we have adapted different Squeak/Smalltalk tools in SQUIMERA which requires 19 classes in total, and around 1,250 SLOC of Smalltalk code.

### 4.1 Building a Multi-language Virtual Machine

In section 3.1, we argue that we need to combine multiple languages in one virtual execution environment in order to be able to reuse Smalltalk tools. To build such a virtual machine, we use interpreter composition. Instead of implementing a programming language on top of another, the VM contains multiple interpreter loops. In the following, we describe how we performed the interpreter composition in a modular way, so that other interpreters can be added, and explain how the different interpreter loops are being executed.

#### 4.1.1 Programming Language Execution

As mentioned in section 3.1, interpreter composition requires some unit that decides when to run code in which interpreter loop. Smalltalk, which acts as the hosting language for other languages, implements co-operative multitasking through *processes* [49, pp. 251–257]. We leverage this mechanism and integrate the execution of a foreign language with a Smalltalk-level process, leaving the decision when to run a foreign interpreter loop up to the Smalltalk scheduler. This will allow us to interact with the Squeak/Smalltalk environment as usual while a program written in a non-Smalltalk language is running. Furthermore, we will be able to see such a language execution process in Squeak/Smalltalk's process browser and interrupt it from there in order to inspect it further.

Inside the VM, such processes are responsible to facilitate the execution of code in the corresponding interpreter loop. They are represented as plugin-specific objects which can also be exposed in the Smalltalk environment. This way, a Smalltalk-level

process can keep a reference to the corresponding language process in the virtual machine. This reference allows us to have more than one foreign language process and can be used to retrieve further information about the execution of the language process in Squeak/Smalltalk including the top frame and the last error.

**Making Stackful Interpreter Loops Interruptible** To be able to suspend and continue a stackful interpreter loop at any point in time, we use RPython stacklets as well as greenlets.

Listing 4.1 contains a minimal example of how a `StackletThread` provided by the `rpython.rlib.rstacklet` package works. We have added `print` statements in the example to highlight the execution flow. The output of this example can be found in Listing 4.2. The `StackletExample` class has a `start` and a `coroutine` method. The `start` method is invoked by the boilerplate functions `main` and `target` which are used by RPython as an entry point. As part of the `start` method, a new `StackletThread` instance is created and then a new stacklet is allocated. For this, the new method of a `StackletThread` takes a callback function which in turn is invoked as soon as the new stacklet starts. In this example, the `stacklet_callback` simply calls the `coroutine` method on the `Example` instance and provides a `handle`. When this `handle` is used in a `switch` call on an instance of `StackletExample`, the program execution continues in the main thread and the initial new invocation returns a fresh `handle` back to the coroutine. With these handles, it is then possible to switch back and forth between the main thread and the coroutine. However, one needs to make sure to always use a valid `handle` and to always only use it once, otherwise the virtual machine may crash with a segmentation fault. Furthermore, it is possible that a `switch` invocation returns an empty `handle`, which usually indicates that a coroutine has terminated. Finally, also `example.start()` finishes and then the program terminates.

We use `StackletThreads` in our virtual machine to implement a runner for our language processes. This `StackletLanguageRunner` class exposes a simple interface that can be used to control the execution of the corresponding language process. Such a process needs to be provided at the time of instantiating a `StackletLanguageRunner`. Then it is possible to `start`, `yield`, and `resume` the execution of the process with the runner. However, `StackletThreads` only work after the VM is translated to machine code. Just like PyPy, RSqueak can be executed in interpreted mode for development purposes. In order to not break the interpreted mode, we have implemented another runner based on so-called greenlets.

With stacklets, we have a generic way to `start`, `stop`, and `resume` stackful interpreter loops in RPython for both, translated and interpreted modes. In a next step, we need to ensure that composed interpreters eventually yield back to Smalltalk, so that the Smalltalk scheduler can choose the next Smalltalk process to execute.

**Ensuring That Interpreter Loops Yield** Whenever code of a foreign language is executed in our virtual machine, the corresponding interpreter is entered and its interpreter loop is executed. Code that runs and returns quickly is unproblem-

**Listing 4.1:** Minimal rstacklet example in RPython

---

```
from rpython.rlib.rstacklet import StackletThread

class StackletExample:
    def start(self):
        print 'main: start'
        self.sthread = StackletThread()
        handle = self.sthread.new(stacklet_callback)
        print 'main: first return from coroutine'
        handle = self.sthread.switch(handle)
        print 'main: second return from coroutine'
        handle = self.sthread.switch(handle)
        assert self.sthread.is_empty_handle(handle)
        print 'main: end'

    def coroutine(self, handle):
        print 'coroutine: start'
        handle = self.sthread.switch(handle)
        print 'coroutine: continue'
        handle = self.sthread.switch(handle)
        print 'coroutine: end'
        return handle

example = StackletExample()

def stacklet_callback(handle, arg):
    return example.coroutine(handle)

def main(argv):
    example.start()
    return 0

def target(driver, args):
    driver.config.translation.continuation = True
    return main, None
```

---

**Listing 4.2:** Output of the `rstacklet` example in Listing 4.1

---

```

main: start
coroutine: start
main: first return from coroutine
coroutine: continue
main: second return from coroutine
coroutine: end
main: end

```

---

atic, because it does not block the execution of other Smalltalk-level processes. Long-running algorithms or even applications that do not terminate, such as web servers, for example, would block other processes if they do not yield back to Smalltalk, because the Smalltalk scheduler is unable to elect the next process to be executed.

Some interpreter loops already have the capability to periodically perform other actions in between executing bytecodes. The Squeak/Smalltalk VM, for example, has an interrupt service routine which frequently checks for mouse and keyboard input [65, p. 72]. With `sys.setcheckinterval()`, it is possible to adjust the interval that determines “how often the interpreter checks for periodic things such as thread switches and signal handlers” [109, p. 23] in Python.

To implement these kind of mechanisms, there often is a simple counter in the interpreter that counts how many bytecodes have been executed. Once a threshold is reached, the periodic tasks are performed. However, having such a counter introduces a performance overhead. In order to minimize this overhead, there are mainly two different strategies, because there are two ways to implement endless routines: with *recursion*, or with *loops*. The former results in frequent *message sends*, the latter in a high number of *backward jumps*.

Nonetheless, it usually is sufficient to increment an interrupt counter and check for interrupts after backward jumps only, to keep the overhead as low as possible. Python and Ruby, for example, have a maximum stack depth or recursion limit which makes recursion impractical for endless routines. Even most Smalltalk interpreters, which do not enforce such a stack depth limit, only check for interrupts after backward jumps and disregard message sends. To detect such a backward jump, only the program counter in the bytecode code loop needs to be observed. If the value of the next *pc* is less than the value of the old counter, a backward jump occurred.

For the Ruby support in SQUIMERA, we need to patch the Topaz interpreter accordingly. First, we introduce a VM-level `InterruptCounter` which we then use in the interpreter loop. In case the loop is not just-in-time compiled, we increment the counter every time a new bytecode is handled by overriding Topaz’s `Interpreter.handle_bytecode` method. However, if the loop is JIT compiled, we only check for interrupts in the `Interpreter.jump` method, and only if a backwards jump is detected. This is an additional optimization to counteract the performance

## 4 Implementation

discrepancy between the interpreted and JIT compiler modes, as the counter is incremented more often when the bytecode loop is not optimized and thus runs slower. Whenever the interrupt counter triggers, the VM switches back to Smalltalk. As a heuristic default value, we use a threshold of 10,000 ticks which is the same value PyPy uses for its interrupt routine. In PyPy, the builtin interrupt mechanism allows us to register a `PeriodicAsyncAction` to periodically switch back to Smalltalk.

### 4.1.2 Preparing Languages for the Smalltalk Debugging Experience

In order to be able to bring the same debugging experience known from Smalltalk to other programming languages, there are two main requirements for language implementations. First, a language needs to be able to identify *unhandled exceptions* on which a debugger is opened in Smalltalk. And second, it needs to be possible to restart execution frames of a language to enable edit-and-continue debugging. Both features are not supported by PyPy and Topaz, nor CPython or Ruby MRI. In the following, we explain how we extended both RPython-based interpreters accordingly.

**Detecting Unhandled Exceptions** A long-running program executed in the way described before may encounter an exceptional situation during its execution. In this case, the corresponding interpreter also needs to yield back to Smalltalk and inform the image of the exception, so that a debugger can then be opened accordingly.

In Python and Ruby, however, the interpreter starts to unroll the stack as soon as an exception is thrown. Then, it might be handled by one of the parent stack frames. But if the exception is not handled at all, the runtime usually fails with a run-time exception instead and the entire process exits with a non-zero status code.

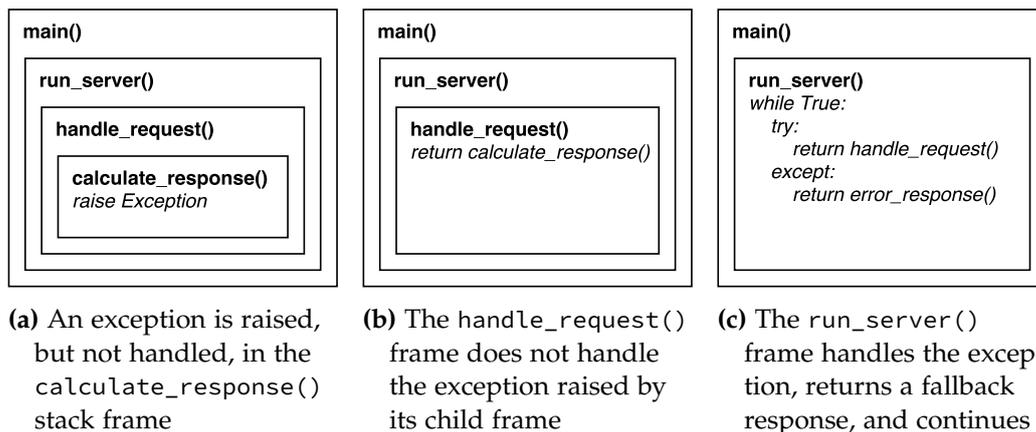


Figure 4.1: Exception handling uses the termination model in Python

Figure 4.1 shows a minimalistic example of how stack unrolling is used to implement the termination model of exception handling in Python. For each method invocation, there usually is a corresponding stack frame. Hence, nested invocations result in nested stack frames. When an exception is raised, the interpreter looks for a corresponding exception handler represented by a special code block in the virtual machine. For that, each frame holds a reference to a linked list of blocks which can handle exceptions, but which are also used to signal other Python statements such as `break`, `continue`, or `finally`. If no block was found for the exception, the frame is removed and the execution continues in its parent frame. This is the case in Figure 4.1a as well as in Figure 4.1b. The interpreter then continues to look for an appropriate block until it finds one or has reached the top frame. In Figure 4.1c, it has found an exception handler in `run_server()` and starts to execute the fall-back behavior. However, at this point in time, the two child frames have already been removed from the call stack. If the `run_server()` frame had not caught the exception, the interpreter would have continued in the `main()` frame and, if unable to find a handler for the exception, finally quit with an error.

In Smalltalk, on the other hand, a debugger is opened on the original stack frame in which the exception was raised instead of unwinding the stack. Only this way users are able to inspect, modify, and resume a running application. Therefore, we need to avoid stack unrolling in Ruby and Python, because it would not be helpful to present a debugger after all stack frames have already been unrolled. This means, unhandled exceptions need to be identified as soon as they are thrown, independently of the language being executed.

---

**Listing 4.3:** Stack frame walking in the PyPy interpreter for SQUIMERA

---

```
def has_exception_handler(self, operr):
    "'True' if the frame or a parent frame is able to handle 'operr'"
    # ...
    frame = self
    while frame is not None:
        # ...
        block = frame.lastblock
        while block is not None:
            # block needs to be an ExceptBlock and able to handle operr
            if ((block.handling_mask & SApplicationException.kind) != 0
                and frame.block_handles_exception(block, operr.w_type)):
                return True
            block = block.previous
        frame = frame.f_backref()
    return False
```

---

The main problem is that neither the Ruby nor the Python interpreters were initially designed to detect unhandled exceptions, because it is not needed to implement the termination model of exception handling. However, it is possible to detect the majority of unhandled exceptions with the following technique: Instead of unrolling the stack, we start to walk the stack frames and search for a corresponding exception handler. This may sound like a straightforward solution. As we will explain later, it is quite complicated to identify unhandled exceptions in practice.

Both, the PyPy and the Topaz interpreters, have methods that are used as soon as an error occurs. In PyPy, we need to patch our unhandled exception checks into `PyFrame.handle_operation_error`, and into `Interpreter.handle_ruby_error` in Topaz. Listing 4.3 shows how the stack frame walking works in the PyPy interpreter. The algorithm begins with the current frame, and starts with its last block. Then, it checks whether the block's `handling_mask` indicates that it handles exceptions. If it is indeed an exception handling block, `block_handles_exception` is called and if it returns `True`, the exception will be handled later on. Otherwise, the next block is inspected in the same way until the first block of the linked list is reached. If this also is unsuccessful, the parent frame is inspected. This is repeated until the top frame is reached. If no appropriate exception handler is found, the function returns `False`. Only in this case, `SQUIMERA` stores `operr` as the last error in the corresponding language process and yields back to Smalltalk. The primitive of a language plugin that is used coordinate the execution of corresponding language processes, the "resume" primitive, checks whether an error occurred during the execution and then fails to inform the Smalltalk environment. In this case, the Smalltalk environment retrieves the last error stored in the language process by using the "lastError" primitive and opens a debugger.

**Checking If an Exception Block Handles a Given Exception** Until now, the implementation to detect unhandled exceptions was straightforward. The problem of reliably detecting unhandled exceptions becomes clearer when we look at the implementation of the `block_handles_exception` function for PyPy: At this point, the frame, the exception handling block, and the exception are known. Unfortunately, blocks do not store the exception that they handle. Instead, an `ExceptBlock` in PyPy only stores a `handlerposition`, a `valuestackdepth`, and a reference to its predecessor. When an exception occurs, it is passed to the first `ExceptBlock`. The block then pops values from the value stack until the given `valuestackdepth` is reached again. After that, it pushes new values including the exception onto the value stack, stores the exception in `frame.last_exception`, and finally returns the `handlerposition` which determines the next instruction in the interpreter loop. This is how the stack unwinding is implemented in the PyPy interpreter. In fact, all `FrameBlocks`, including a `FinallyBlock` and a `LoopBlock`, work like this, which is why the mechanism is implemented in such a generic way. Consequently, we have to analyze the call stack and the frames' value stacks if we want to avoid major design changes in the PyPy interpreter.

**Listing 4.4:** A simple exception handling example in Python

---

```

try:
    print 1/0
except ZeroDivisionError:
    print 42

```

---

**Listing 4.5:** Disassembling Listing 4.4 with Python's disassembler `dis`


---

2		0 SETUP_EXCEPT	13 (to 16)
3		3 LOAD_CONST	0 (1)
		6 LOAD_CONST	1 (0)
		9 BINARY_DIVIDE	
		10 PRINT_ITEM	
		11 PRINT_NEWLINE	
		12 POP_BLOCK	
		13 JUMP_FORWARD	22 (to 38)
4	>>	16 DUP_TOP	
		17 LOAD_NAME	0 (ZeroDivisionError)
		20 COMPARE_OP	10 (exception match)
		23 POP_JUMP_IF_FALSE	37
		26 POP_TOP	
		27 POP_TOP	
		28 POP_TOP	
5		29 LOAD_CONST	2 (42)
		32 PRINT_ITEM	
		33 PRINT_NEWLINE	
		34 JUMP_FORWARD	1 (to 38)
	>>	37 END_FINALLY	
	>>	38 LOAD_CONST	3 (None)
		41 RETURN_VALUE	

---

## 4 Implementation

Listing 4.4 shows a simple example for exception handling in Python. First, the program attempts to print `1/0` which raises an `ZeroDivisionError`. This error is then caught by the *try-except* block and finally, `42` is printed to `stdout` instead. Listing 4.5, on the other hand, demonstrates how the program is executed in the interpreter loop: In the first line, an exception block is created and added to the frame. Its `handlerposition` points to line 16. Then, two constants are loaded onto the value stack, the numbers 1 and 0. The `BINARY_DIVIDE` bytecode performs the division. Normally, it pops the first two values from the value stack, divides the values, and pushes the result back onto the value stack. Afterwards, the result and a newline are printed to `stdout`. Since the exception block would no longer be needed, it will be popped from the frame. After that, a jump to instruction 38 is performed, which jumps over the exception handling instructions. Finally, the program terminates after pushing `None` onto the value stack and returning it which is the default behavior of a top frame.

But, since the program tries to divide 1 by 0, an exception is raised in `BINARY_DIVIDE`. This causes the interpreter to look for an `ExceptBlock`, which is then executed and which redirects the execution flow to instruction 16. The `DUP_TOP` bytecode duplicates the top value on the stack, which is the exception. Then, the name of the exception of the `ExceptBlock` is loaded. In line 20, the type of the raised exception is compared to the exception handled by the *try-except* block. In case the exceptions do not match, a jump to instruction 37 would occur, and the interpreter would fail, because of an unhandled exception. In the example, the check is successful and the top three values are popped from the value stack again to clean it up. Then, instruction 29 loads `42` onto the stack again which is finally printed followed by a newline, followed by a jump to 38.

In case of the example in Listing 4.4, we perform bytecode analysis as soon as `BINARY_DIVIDE` throws an error. First, the interpreter has to look for the closest `ExceptBlock`. Then, we need to find out to which instruction in the bytecode the block will jump. In all cases similar to the example, we expect to find the `DUP_TOP` bytecode which is used to duplicate the exception on the value stack. This duplicate and an exception name is then consumed by `COMPARE_OP`. Therefore, we need to look for the instruction after the `DUP_TOP` which is expected to be the `LOAD_NAME` bytecode. In the instruction call, we find the index of the name which we can then use to look up the name of the exception that the block handles. Finally, we only need to compare the raised exception with this exception name to check if the block will handle it.

Nevertheless, there are many other ways to use *try-except* block in Python. The example in Listing 4.6 demonstrates a few of them. *Try-except* blocks can be nested, there can be multiple *except* statements in one *try-except* block, and there is a way to specify to simply catch all exceptions. Additionally, the instance of an exception can be passed into an *except* case. Furthermore, one can not only specify one, but multiple exceptions per *except* statement using a Python `tuple`. And, of course, variables and functions can be used to dynamically specify which exceptions to catch. The `exceptions_to_catch` variable in the example is randomly set to either `ZeroDivisionError` or `(ZeroDivisionError, IndexError)`, which causes

**Listing 4.6:** An example of other exception handling techniques in Python

---

```

exceptions_to_catch = ZeroDivisionError
if bool(random.getrandbits(1)):
    exceptions_to_catch = (exceptions_to_catch, IndexError)

try:
    try:
        list()[1]
    except exceptions_to_catch:
        print 'inner fallback'
except IndexError as e:
    print 'outer fallback'
except:
    print 'catch-all fallback'

```

---

the program to randomly print “inner fallback” or “outer fallback”. Furthermore, there are many other ways to use try-except blocks in Python, for example by using metaprogramming. For SQUIMERA, we have implemented detection algorithms for the most common ways to catch exceptions. However, it is impossible to identify unhandled exceptions reliably due to the ability to dynamically specify exceptions with functions or metaprogramming.

**Listing 4.7:** Example of an exception being masked by a Python builtin

---

```

class MyClass(object):
    def __getattr__(self, name):
        if name == 'x':
            raise AttributeError

instance = MyClass()
getattr(instance, 'x', 42)

```

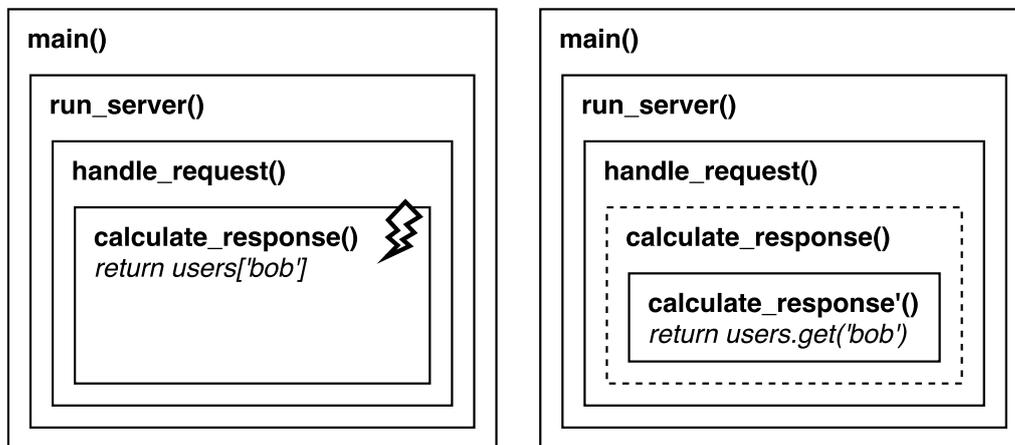
---

Additionally, Python builtins also need to be considered, because they can mask exceptions. Listing 4.7 shows an example for that. `MyClass` implements the method `__getattr__` which is internally used by Python to retrieve attributes of its instances. If `getattr(instance, 'x')` is called, an `AttributeError` exception is raised to indicate that the instance does not have an attribute called `x`. The Python builtin `getattr`, however, also accepts an optional default parameter which is returned in case of an `AttributeError`. Hence, `getattr(instance, 'x', 42)` returns `42` in the example and suppresses the exception, just like `next(iter([]), 42)` which is another example for the same problem. This means, to detect unhandled exceptions,

the interpreter must not only check for corresponding `ExceptBlocks`, but also for potential builtins that may handle a given exception type internally. And not only a list of these kind of builtins is needed to check for each exception type, the algorithm also needs to perform further bytecode analyses to figure out whether the builtin was called with or without a default value. `SQUIMERA` supports both, the example with the `getattr` builtin and the example with the `next` builtin. Nonetheless, its unhandled exception detection still does not cover all cases. We will discuss further limitations of this approach in section 5.5.

**Adding Support to Restart Stack Frames** To provide a Smalltalk-style debugging experience, it is not only necessary to detect unhandled exception. In addition, it also needs to be possible to modify a program at run-time. The Python debugger module `pdb` uses Python's global trace function to intercept the execution of a program at run-time [109, pp. 75–76]. Even though, the debugger can be used to, for instance, change values of variables, the execution frame can only be inspected with Python's `inspect` module and its attributes are mostly read-only. A frame's `f_code` attribute, for example, is read-only and holds the code object currently being executed. But to make the Smalltalk debugger work the same way for Python programs, it is required to be able to swap this code object with a new one and to restart the frame afterwards.

As we demonstrate now, it is possible to extend the PyPy interpreter so that it supports replacing the code in a frame and to restart it. First, we need to ensure that a `PyFrame` object keeps all information that was used to create it. This way we can later swap parts of this information with user-provided data and create a new `PyFrame` object as a replacement.



(a) Stack frames of a program at the time an `KeyError` is thrown and a debugger is opened

(b) Stack frames after the user modified the `calculate_response()` method and restarted the frame

**Figure 4.2:** Example of how PyPy stack frames are restarted in `SQUIMERA`

Figure 4.2a shows an example debugging scenario: A debugger is opened, because “bob” was not found in the users dictionary which raised a `KeyError` in Python. When the user now modifies code in the debugger and saves it, the new code as well as the selected frame are saved in the VM as restart information. At this point, the corresponding language process is not executed by the Smalltalk scheduler, because the debugger is attached to it. When the user then decides to proceed with the execution, the language process will be scheduled for execution soon after. Before such a process is resumed, the resume routine checks if a user has set restart information for a specific frame and then throws a `RestartException` at the current point of execution in the bytecode loop. This exception is then propagated through the call frame stack until the designated frame is found. This frame finally extracts the restart information, such as the modified code, from the exception and creates a new frame accordingly, which is then executed inside, but instead of the original frame. Similar to Smalltalk, side effects are not rolled back. Hence, if a method, for example, deleted a file, restarting the frame will not restore the file again.

The same approach can be applied the Topaz interpreter for Ruby. Nonetheless, additional work is needed, because information on the current execution frame is normally not exposed in Ruby. This means, the VM also needs to be able to expose Topaz’s `Frame` and `BuiltinFrame` objects. For this, we implement a wrapper class that inherits from Topaz’s `W_BaseObject` which the virtual machine then can use to expose frames just like any other Ruby object in the Smalltalk environment.

### 4.1.3 Abstracting from Different Foreign Languages

As briefly described with Figure 3.2, there is a VM-level plugin for each foreign language, which are part of the `rsqueakvm.plugins` package of RSqueak/VM. These plugins expose a consistent API that is used to communicate with the Squeak/Smalltalk environment. Figure 4.3 illustrates the VM architecture of SQUIMERA. To enable code sharing between different foreign language plugins, we introduce an abstract `ForeignLanguagePlugin` class. This class implements all primitives needed for the communication with the image. Moreover, it knows how to pre-load special Smalltalk classes that are used by the VM to expose foreign objects in the image. Lastly, it has a number of abstract methods that need to be implemented by each subclass of this plugin.

Then, there is a `ForeignLanguageProcess` class which is internally used to represent language processes. Therefore, it knows how to switch between Smalltalk and a foreign language, and can store the result or an error of the language execution. Since the execution of a foreign language itself is language-specific, the class also implements abstract methods that need to be overridden by each language that is integrated into SQUIMERA.

When the virtual machine is translated, a language process is executed by a `StackletLanguageRunner`, in interpreted mode and for development purposes by a `GreenletLanguageRunner`, both of which are also part of the `foreign_language` package. In addition, the package also provides the two abstract classes `ForeignLanguageObject` and `ForeignLanguageClassShadow`. Both classes implement

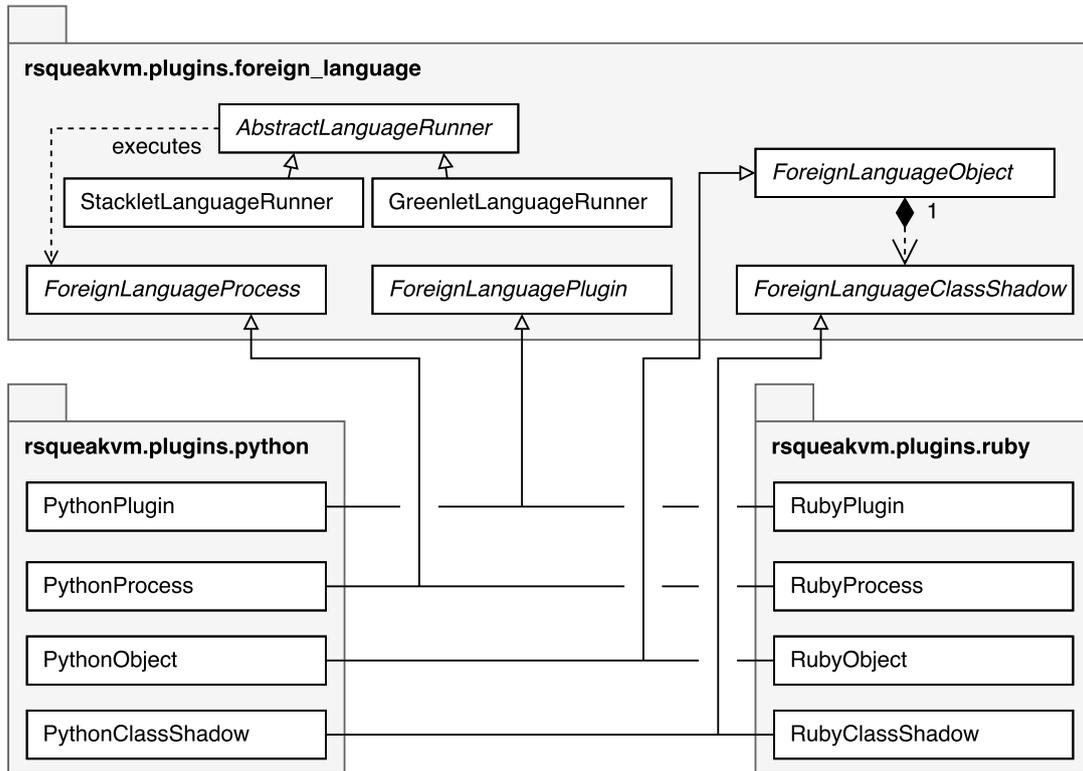


Figure 4.3: Architecture of SQUIMERA's virtual machine

common behavior as well as abstract methods which are again overridden by corresponding subclasses for each language. These subclasses are used to expose foreign language objects in the Squeak/Smalltalk environment and, for example, implement the method lookup. First, the lookup is performed in the corresponding Smalltalk class, in `PythonObject` for Python for example. This allows us to later implement Smalltalk methods that bridge between Smalltalk and the language, so that foreign language objects can provide the same meta-object protocol as native Smalltalk objects which is then used by different tools. After that, the method lookup continues in the corresponding language implementation. We will explain the method lookup in more detail in subsection 4.2.1. If a method was found, however, the class shadow either returns the `CompiledMethod` in case of a Smalltalk method or builds and returns a fake Smalltalk `CompiledMethod` which will activate the “send” primitive of the appropriate language plugin.

For each language integration, we have added a dedicated sub-package in the `rsqueakvm.plugins` package. The `rsqueakvm.plugins.python` package contains subclasses of all abstract classes from the foreign language package and only implements language specifics. With that, we mean not only the specialized subclasses, but also other components such as the way an object space is set up for the interpreter, the different patches for the interpreter, as well as utility functions that can convert primitive data types of the language it executes to native Smalltalk objects.

Moreover, a `PythonProcess` not only holds the information for its execution provided by the user. Each Python process also stores an execution context which contains the stack frames for the program being executed. When a process is scheduled, the execution context will be used by the interpreter for the execution. This way it is possible to have multiple programs running at the same time, because each process uses its own execution context.

As Figure 4.3 suggests, the Ruby plugin is implemented in exactly the same way as the Python plugin and additional language integrations can be added similarly.

**The Virtual Machine Primitives** Table 4.1 lists the common primitives of a `ForeignLanguagePlugin`. The “eval” primitive is the entry point for the execution of a foreign language. Its arguments are passed to a language-specific method that is responsible for creating a new language process to which the primitive then yields. For Python, not only the Python source code is needed, but also a filename as well as a string which describes the execution mode. This string can either be *eval*, *exec*, or *single*. For Ruby, only the source code and a filename are required to create a language process. If the process yields, the primitive returns a fake Smalltalk frame which will resume the language process the next time it is scheduled. For this, the “resume” primitive is used. In case the process completes, the primitive returns a fake Smalltalk frame that returns the result of the process instead.

When a user decides to send a message to a foreign object, the “send” primitive is called with the foreign object as the `receiver` and a method `selector` for the message name. Similar to the “eval” primitive, the execution of a message send is also coordinated as part of a dedicated language process which can yield and then be resumed by the “resume” primitive. This way, long-running message sends are handled in exactly the same way long-running programs are executed by the “eval” primitive.

In case the “resume” primitive fails, the Smalltalk environment can use the “lastError” primitive to retrieve the error from a language process. This error can either be another foreign language object representing the error, or a native Smalltalk object depending on the language-specific implementation. However, this information can be used when a debugger window is opened on the language process.

**Table 4.1:** General set of primitives of a `ForeignLanguagePlugin`

Primitive name	Arguments
eval	<i>language-specific</i>
resume	<code>languageProcess</code>
send	<code>receiver, selector, args</code>
lastError	<code>languageProcess</code>
topFrame	<code>languageProcess</code>
restartSpecificFrame	<i>language-specific</i>
asSmalltalk	<code>languageObject</code>

In addition to the error, the Squeak/Smalltalk debugger is able to retrieve the frame in which the error occurred using the “topFrame” primitive. This primitive also takes a language process as an argument and returns the top frame from the process’ execution context wrapped as a Smalltalk object of the language. In Topaz, a `RubyObject` with a reference to a Topaz frame object is returned, in Python it is a `PythonObject` with a reference to a `PyPy PyFrame`.

The Smalltalk debugger can retrieve further information, such as the current line of execution and the corresponding source code from these frame objects. To restart a specific frame, the debugger uses the “restartSpecificFrame” primitive which takes a frame as well as further language-specific values as arguments. This primitive stores the restart information inside the VM which will cause the frame to be restarted as described before in subsection 4.1.2.

Finally, the “asSmalltalk” primitive allows the conversion of primitive data types of a foreign language to native Smalltalk objects. This primitive is used in the `asSmalltalk` method that all `ForeignLanguageObjects` implement. When the message is sent to, for example, a Ruby `Fixnum`, the corresponding Smalltalk `Integer` is returned. Python `tuples` and Ruby `Arrays` are recursively converted to Smalltalk `Arrays` with Smalltalk items. The main purpose of this primitive is to enable the language integration, so that the tools can be reused. As we will explain later, this facility also allows the reuse of software libraries and frameworks written in different languages.

## 4.2 Adapting Smalltalk Tools for Other Languages

Now, that we have built an execution environment that is capable of executing different programming languages, we need to extend the Smalltalk environment, so that we can leverage these new features.

Figure 4.4 shows the architecture of the different components `SQUIMERA` brings into a Squeak/Smalltalk environment, so that its tools can be used for other integrated languages. The `ForeignLanguage-Core` package contains the key classes which need to be subclassed for each additional language. The `ForeignLanguage` class does not implement any behavior on instance-level. Instead, it only provides static methods for the communication with the corresponding VM plugin and language-specific helper functions to support the different tools.

All objects from foreign languages are exposed by `ForeignLanguageObject` subclass instances. Hence, the `ForeignLanguageObject` class implements behavior that is common across different language integrations as well as abstract methods that require a language-specific implementation.

In order to perform the execution of a foreign language, the abstract class `ForeignLanguageProcess` is used. Objects of this class represent a single language execution inside the virtual machine and can be suspended, resumed, or terminated just like any other Smalltalk process. Moreover, appropriate VM primitives can be used to retrieve the last error or the top foreign frame of the corresponding execution context.

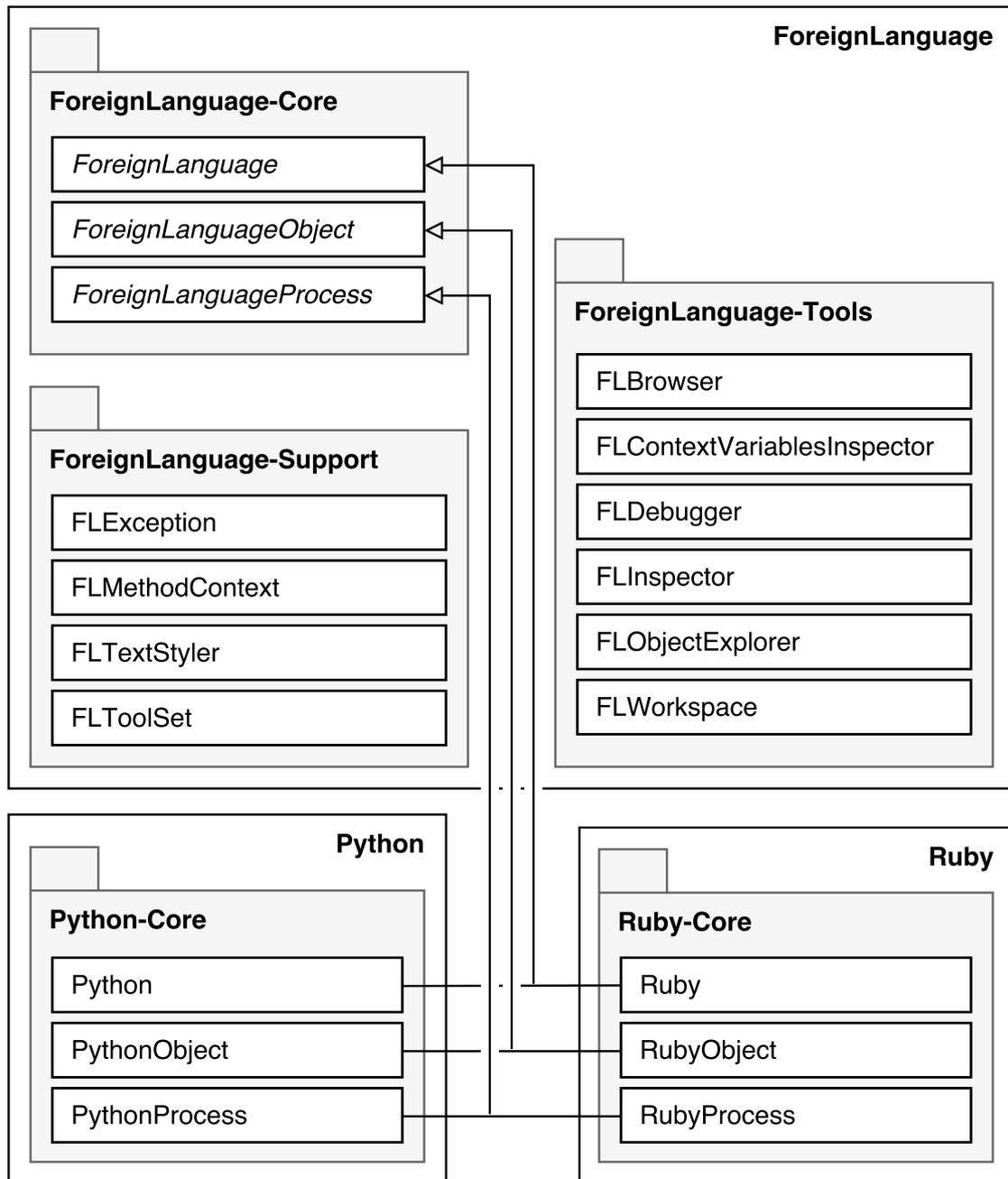


Figure 4.4: Architecture of SQUIMERA's development environment

## 4 Implementation

Further, the `ForeignLanguage-Support` package contains different helper classes used by adapted tools or the language integration. An `FLException` is thrown as soon as the “resume” primitive fails and indicates an exception in a foreign language, which in turn then opens a debugger. Since execution frames are implemented as `MethodContexts` in Squeak/Smalltalk, we introduce a `FLMethodContext` which will be used by the debugger to reference foreign frames. The `FLTextStyle` is used by different tools that display code for syntax highlighting purposes. Moreover, the `FLToolSet` class implements the interface used by Squeak/Smalltalk to request the different tools and makes all adapted tools the default.

All tool adaptations are part of the `ForeignLanguage-Tools` package and subclass from the corresponding Squeak/Smalltalk tool.

Finally, there are the two packages `Python-Core` and `Ruby-Core` which, similar to the VM plugins, are implemented in a consistent way. Both introduce subclasses for the three classes of the `ForeignLanguage-Core` package which is everything needed to be able to work with these languages in the Smalltalk environment.

### 4.2.1 Integration on Language-level

Before being able to adapt any tools, we need to be able to interact with an integrated language from Smalltalk, because Smalltalk acts as the hosting language in SQUIMERA. The most basic way of doing this is via a special evaluation method for a particular language. Therefore, we need to implement the `ForeignLanguage>>eval:` method, which uses a specialized method to call out to the “eval” primitive of the corresponding language plugin in the virtual machine. With this, we are able to execute, for example, `Python eval: '[1, 2]'` or `Ruby eval: '[1, 2]'` which return either a `PythonObject` or a `RubyObject`.

After that, we need to provide the ability to send messages to these objects in order to interact with them. For this reason, we have tweaked the method lookup in the virtual machine as mentioned in subsection 4.1.3. We now explain how we have mixed the different language semantics. On each message send to a foreign object, the lookup starts in the corresponding Smalltalk class, `PythonObject` for example. This allows us to implement and call Smalltalk methods which we later need for adapting tools. If no Smalltalk method is found, the lookup continues in the foreign language. Since Smalltalk uses selectors as method names, we need to derive method identifiers for Python and Ruby from them. This can be done by treating Smalltalk selectors as strings and copying them until the first colon. The method name derived from the selector `#copy:from:into:` is, for example, just “copy”, but it still can be used to perform a message send with three arguments.

However, there are language-specific details that cause different problems. In Python, for instance, objects have attributes. A data attribute usually represents an instance or class variable, but methods are also attributes. If we send the message `insert` to a Python `list`, for example, it is clear that we are dealing with a method on `list`. Nonetheless, it is not clear whether we want to call the method `insert` or just retrieve the reference to that method.

Moreover, Python uses a special method identifier `__call__` for its callables, which are all objects that can be called, including methods as well as user-defined objects. We decided to use this identifier to describe the only message that invokes a method call in Python. Therefore,

```
(Python eval: '{"x" : 800}') get
```

retrieves a reference to the bound method `get`, while

```
(Python eval: '{"x" : 800}') get __call__: 'y' and: 600
```

actually calls that method which attempts to return the value for `y`, but instead returns the default value 600 since `y` is not in the dictionary.

---

**Listing 4.8:** A Python function to check if code is a Python expression

---

```
def is_expression(code):
    try:
        compile(code, '<string>', 'eval')
        return True
    except SyntaxError:
        return False
```

---

Another issue we encounter is that there is a difference whether code is *evaluated* or *executed* in Python. The former only works if the code represents a single Python expression and returns the result. It does not work for assignments, imports, or anything else. In these cases, the source code needs to be executed instead and `None` is always returned as the result.

To avoid implementing an algorithm that performs a check correctly in Smalltalk, we leverage the Python compiler and use the Python function shown in Listing 4.8 instead. For convenience reasons, we also implement a Smalltalk method `Python>>isExpression:` which calls out to the Python implementation and returns a Smalltalk `Boolean`. With this, we can implement `Python>>eval:`, so that it correctly either evaluates or executes the provided source code.

The Ruby integration works similarly, but another problem becomes more prominent: Since we are using Smalltalk selectors for method identifiers and because it is possible to call both Smalltalk and foreign language methods, we have to deal with name clashes. Sending the message `inspect` to a Smalltalk object opens an inspector on the object [48, p. 144]. However, the same message retrieves the string representation of objects in Ruby. Therefore, there needs to be a way to call both. It is best practice to not use method selectors that start with an underscore in Smalltalk, even though selectors starting with underscores are allowed by the language. For this reason, we decided to implement a fallback lookup in case of

such a method identifier. Hence, if we want to send `inspect` to a `RubyObject` in Ruby, we can use `(Ruby eval: '[1, 2]') _inspect` to avoid calling the Smalltalk method and to call the Ruby method instead.

---

### Listing 4.9: `printOn:` implementation for Python objects

---

```
PythonObject>>printOn: aStream
  aStream
    nextPutAll: 'Python ';
    nextPutAll: (Python builtins str __call__: self) asSmalltalk
```

---

---

### Listing 4.10: `printOn:` implementation for Ruby objects

---

```
RubyObject>>printOn: aStream
  aStream
    nextPutAll: 'Ruby ';
    nextPutAll: self _inspect asSmalltalk
```

---

At this point, we are now able to implement the Smalltalk method `printOn:` for each, the `PythonObject` and the `RubyObject` classes, as shown in Listing 4.9 and Listing 4.10. These methods are used by different tools to gather the string representation of an object in Smalltalk.

The implementation for Python objects first puts the string `'Python_'` onto `aStream` which helps users to understand of which language an object is. Then a shortcut to the `builtins` module is used for convenience and caching reasons. It then sends `str` to retrieve the Python `str` type from the `builtins` module, which is then called with the object of interest as an argument. The equivalent to this in pure Python is `str(self)`. The result is always a Python string which can be converted by the “`asSmalltalk`” primitive to a Smalltalk `ByteString`. This string is lastly also printed onto the stream. As a result, we can now evaluate `Python eval: 'object()'` in a workspace as part of a `printIt` which will then display `Python <object object at 0x000000011bfd8588>` for example.

The Ruby version works similar, but sends `_inspect` to the object which returns its Ruby string representation and which is then again converted into a Smalltalk string object.

After the languages have been integrated, so that we can evaluate code and send Smalltalk messages to their objects, we can override more Smalltalk methods from the meta-object protocol which are used by the tools and start adapting them.

#### 4.2.2 Adapting Squeak/Smalltalk's Workspace and Inspector

The first tools that we adapt are the Smalltalk workspace and inspector. Like many other tools that display text, the workspace uses a `TextEditor` view for its code editing capabilities. This `TextEditor` dispatches evaluation requests on the model, which is the `Workspace` class itself. Therefore, we can introduce a new subclass `FLWorkspace` and all we have to do to redirect evaluation requests to the corresponding `eval:` method for either Python or Ruby, is to implement a method called `evaluateExpression:`.

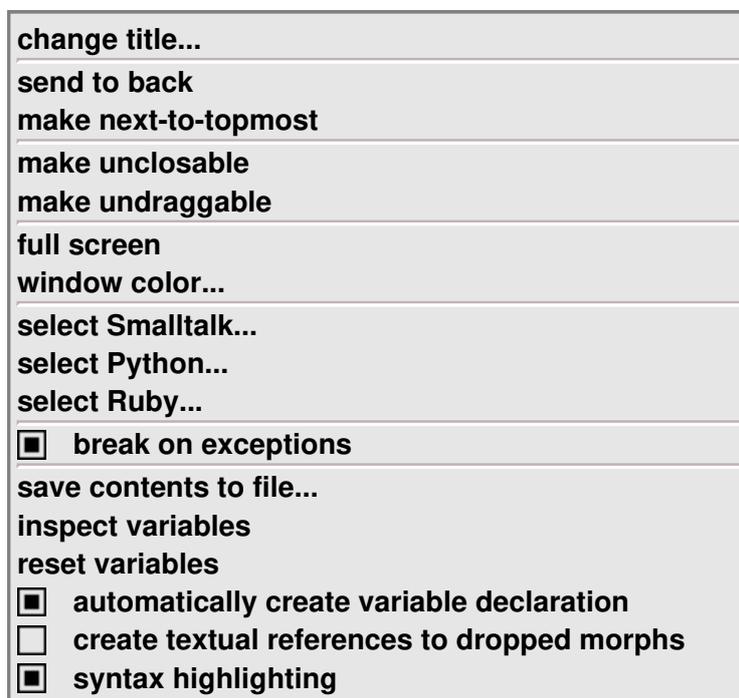


Figure 4.5: Languages can be switched in the workspace's menu

Next, we add menu hooks which change an instance variable `languageSymbol`. This variable can either be `#Smalltalk`, `#Python`, or `#Ruby`, which are Smalltalk symbols we can use to look up the corresponding language class. With this, an instance of `FLWorkspace` in Smalltalk mode can easily be turned into a Ruby workspace or into a Python workspace and then back into a workspace for Smalltalk. In the same way, we add the ability to enable or disable `breakOnExceptions`, which can be passed as an additional argument when calling an "eval" primitive. Figure 4.5 shows the resulting context menu of a `FLWorkspace` instance.

## 4 Implementation

Similar to the workspace, we adapt the inspector tool. We start with a new subclass `FLInspector`. In order to ensure that foreign language objects are opened in the adapted tool, we implement `inspectorClass` on the `ForeignLanguageObject` class. This method is shared across all integrated languages and has to simply return the class of the inspector to open. Additionally, this also implies that Smalltalk objects will still be opened in the original inspector, so a `FLInspector` will only be opened on foreign objects.

In addition, we override `fieldList` in `FLInspector` as well as `selection`. The former method is responsible to provide the elements for the list displayed in the inspector, the latter is used to retrieve the content for the widget on the right for the element selected in the list. The original `Inspector>>fieldList` dispatches the `allInstVarNames` message on the object's class in Smalltalk. In Python and Ruby, it makes more sense to dispatch this message on instances, therefore, we can implement `FLInspector>>fieldList` with

```
↑ #('self' 'all elements'), object allInstVarNames.
```

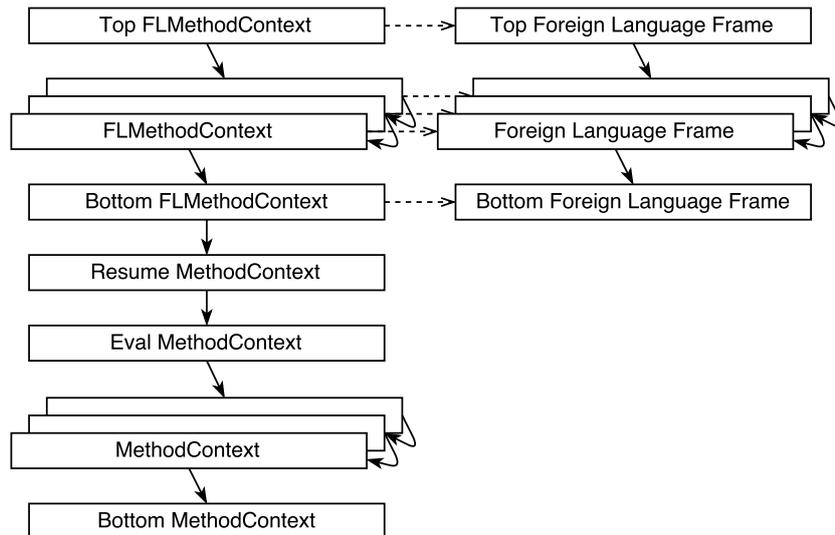
To request the value of an instance variable, we use `instVarAt:` which we also implement and dispatch on the foreign language objects. Finally, to make the workspace-like widget on the bottom work, we again implement an `evaluateExpression:` method, this time on `FLInspector`. Since code is evaluated in a specific context and on a given receiver, additional information needs to be passed to the specific language implementation. Therefore, we dispatch these evaluation requests on the corresponding language class by calling `evaluateExpression:in:to:`, which we have implemented for both, Ruby and Python objects.

### 4.2.3 Adapting Squeak/Smalltalk's Debugger

Compared to the workspace and inspector tools, the Smalltalk debugger is a much more complex tool, as it has to deeply integrate with the underlying run-time environment. On the other hand, it is probably the best tool to demonstrate the advantages of our approach, for exactly the same reason.

We again start with a subclass of Squeak/Smalltalk's `Debugger` class. Since we intend to retain the debugger's original behavior, as mentioned in section 3.1, we can use our new `FLDebugger` as a drop-in replacement. To ensure that the Smalltalk environment uses the debugger adaptation by default, we implement the debugger-related methods on the `FLToolSet` class, for example the one used when an user interrupt occurs.

In addition, the debugger needs to be opened when an unhandled exception is thrown in a foreign language process. Therefore, `SQUIMERA` has to check for an exception using the "lastError" primitive whenever the "resume" primitive of a language plugin failed and then open the debugger accordingly. Since the debugger operates on `MethodContext` instances, we add a subclass `FLMethodContext` which can be used to represent frames of a foreign language in our debugger.



**Figure 4.6:** FLMethodContext objects representing foreign frames are stacked on top of Smalltalk method contexts

When the debugger is opened on a MethodContext, it first checks if the context references the special `vmResume` method the VM uses to run and resume foreign language processes. Only in this case, it manipulates the chain of contexts as illustrated in Figure 4.6. The “`topFrame`” primitive is used to retrieve the top foreign frame from the language process of interest. As a loop walks down the call stack, frame by frame, new FLMethodContext objects are generated for each frame and linked accordingly. Then, the sender of the bottommost FLMethodContext representing the bottommost foreign frame is set to the resume context on which the debugger was opened. This way, we are able to continuously stack foreign frames on top of Smalltalk contexts. Finally, the debugger continues with its initialization on the topmost FLMethodContext.

By overriding `messageIconAt:`, we can instruct the debugger to display either a Python or a Ruby icon depending on the referenced foreign frame of an FLMethodContext object.

Additionally, we can override the behavior for each button as they call a designated method each. The `restart` method, for instance, is called when clicking the “restart” button. In our debugger’s case, it has to trigger the “`restartSpecificFrame`” primitive of the language plugin with appropriate arguments.

To make the debugger display the correct code, we override `selectedMessage` and let it return the corresponding source code in case a FLMethodContext is selected. For this, `sourceCode` is overridden in FLMethodContext and retrieves the source code by dispatching a language-specific message `getSource:` with the frame as argument.

The method `contents:notifying:` is used by the debugger when the user decides to modify and save code in the editor. Therefore, we override it and call out to the

“restartSpecificFrame” primitive again, which patches and restarts the frame with the new source code.

Finally, we override `expandStack` in order to be able to swap the receiver inspector with the `FLInspector` as well as context variable inspector with a new `FLContextVariableInspector` that we implemented for foreign languages. Both inspector units are automatically embedded in the debugger.

### 4.2.4 Reusing a Python Library for Syntax Highlighting

The different tools that we have adapted all display source code somehow in a `TextEditor` view. But since we are able to switch between languages to which the tools adapt, we also want to provide syntax highlighting across all supported languages.

The syntax highlighting for Smalltalk code is provided by the `Shout` package in `Squeak/Smalltalk`. The `SHTextStyler` class defines the interface that is used by all tools for syntax highlighting purposes, while the `SHTextStylerST80` implements the styling of Smalltalk-80 source code. However, this class uses the Smalltalk parser `SHParserST80`, which means that we would need to implement our own parser and styler for each, Python and Ruby.

Instead, we use this opportunity to demonstrate how `PYGMENTS`, a syntax highlighting library written in Python with support for over 300 different programming languages [21], can be reused in `SQUIMERA`. Therefore, if we are able to integrate `PYGMENTS` as a syntax highlighter in `Squeak/Smalltalk`, we can immediately style Python, Ruby, and many other languages without having to implement different parsers.

Our `FLTextStyler` class inherits from `SHTextStylerST80` rather than from `SHTextStyler`, because most tools directly rely on the specialized version. Then, we override all methods in all adapted tools that are responsible for instantiating their text styler and swap `SHTextStylerST80` with `FLTextStyler`.

#### Listing 4.11: How `SQUIMERA` reuses `PYGMENTS` for syntax highlighting

---

```
FLTextStyler>>highlight: aText lexer: aLexer
  Python vmSpeaksLanguage ifFalse: [ ↑ aText ].
  ↑ (Python pygmentsHighlight
    __call__: aText string
    lexer: aLexer
    formatter: Python htmlFormatter) asSmalltalk asTextFromHtml
```

---

`PYGMENTS` API exposes the method `highlight` which takes code, a lexer, and a formatter as arguments. It further provides lexers for all supported languages, as well as different formatters. Listing 4.11 shows how we use all of this in `SQUIMERA`. The `highlight:lexer:` is used by the `FLTextStyler` to style Smalltalk `Text` objects.

The correct lexer is retrieved from the corresponding language class and then provided as the `aLexer` argument. First, it checks whether Python is supported, otherwise it returns the unchanged text again which means that no styling is applied. If Python is supported, the `pygments.highlight` method is called with appropriate arguments. As the formatter, we use PYGMENTS' `HTMLFormatter`. We turn the resulting Python string into a Smalltalk `ByteString` with `asSmalltalk` and finally convert the HTML string to a Smalltalk `Text` object. This is all we have to do to integrate PYGMENTS in SQUIMERA, so that it provides syntax highlighting in all adapted tools and for all supported languages.

### 4.3 Summary

Our prototype system SQUIMERA is an implementation of our approach and combines Python, Ruby, and Smalltalk. Its virtual machine contains an interpreter for each language. Support for Ruby and Python is implemented as part of vm-level plugins which makes it easy to add more languages. SQUIMERA enables the ability to call from Smalltalk to other languages, which are then executed as part of a Smalltalk-level process and by using Squeak/Smalltalk's cooperative scheduling mechanism. Additionally, the SQUIMERA environment provides different tool adaptations including a workspace, inspector, and debugger that work similarly across all supported languages.

## 5 Evaluation

In chapter 4, we explained how SQUIMERA is implemented. Next, we want to show how the SQUIMERA system can be used and demonstrate that the tool adaptations work on Python and Ruby objects the same way they work on Smalltalk objects. For this, we have prepared different use cases which allow us to compare the interaction with the tools when working with different languages.

First, we show how the Squeak/Smalltalk environment can offer live exploration tools that work consistently across all languages. Then, the interaction with the debugger on programs and objects from all languages is illustrated and compared. Finally, we give examples of how external software libraries and frameworks can normally be used in Squeak/Smalltalk and how SQUIMERA provides a more convenient way.

### 5.1 Live Object Exploration

A Smalltalk-80 environment, as described in section 2.2, offers various tools for live data and object exploration. The Smalltalk workspace works similar to a code editor, and also allows interactive code evaluation. The Smalltalk inspector can be used to examine the internals of an object.

Figure 5.1a shows a Squeak/Smalltalk workspace which is used to interactively evaluate an application-specific code snippet. First, a new `DataStack` object is instantiated and stored in a variable `ds`. Then, different elements are pushed onto the stack. After that, the developer performs a `printIt` on `ds` to retrieve the object's string representation which shows the result of evaluating all previous expressions.

Next, an `inspectIt` is performed on `ds` which opens an inspector window, as shown in Figure 5.1b. This displays the object's string representation, but also lists an instance variable `linkedList` which the developer may inspect further. Nonetheless, this already revealed that objects of the `DataStack` class use a linked list to manage the elements internally.

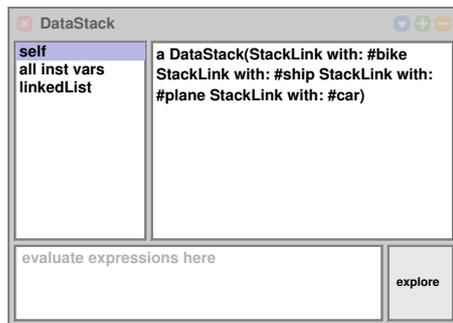
In Figure 5.1c, the `pop` method was sent to the inspected object. This can be done by using the widget on the bottom of the inspector interface which works similar to a workspace. Additionally, the `self` keyword is bound to the inspected object. As a result, the last entry that was pushed onto the stack is returned and displayed as part of the `printIt`. Moreover, the string representation has also been updated to reflect the change of the object. Since an entry has been popped off the `DataStack`, it only contains three remaining entries. This is a simple example to show how the inspector can provide live, immediate feedback on Smalltalk objects.

```

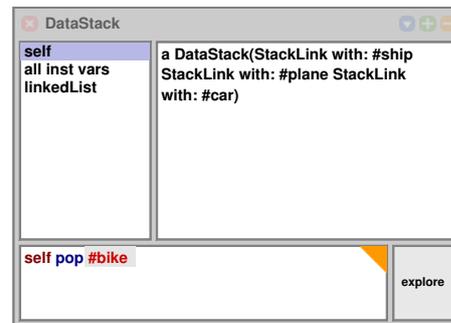
Workspace
| ds |
ds := DataStack new.
ds push: #car; push: #plane; yourself.
ds push: #ship; push: #bike; yourself.
ds a DataStack(StackLink with: #bike
StackLink with: #ship StackLink with: #plane
StackLink with: #car)

```

(a) Iteratively trying out an application-specific DataStack class



(b) After opening the inspector tool



(c) After calling the pop method

**Figure 5.1:** Inspecting an application-specific object in Squeak/Smalltalk

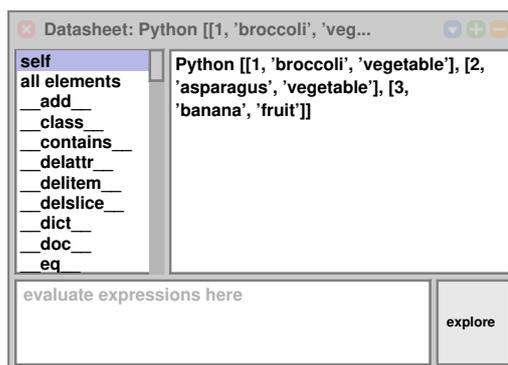
```

Python Workspace
from pattern.db import Datasheet, INTEGER, STRING, uid

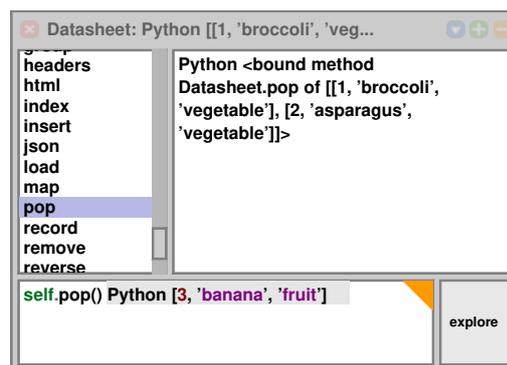
ds = Datasheet(rows=[
    [uid(), "broccoli", "vegetable"],
    [uid(), "asparagus", "vegetable"],
    [uid(), "banana", "fruit"],
], fields=[
    ("id", INTEGER), ("name", STRING), ("type", STRING)
])
ds Python [[1, 'broccoli', 'vegetable'], [2, 'asparagus', 'vegetable'], [3, 'banan

```

(a) Iteratively trying out an example of the PATTERN module



(b) After opening the inspector tool



(c) After calling the pop() method

**Figure 5.2:** Inspecting an application-specific Python object in SQUIMERA

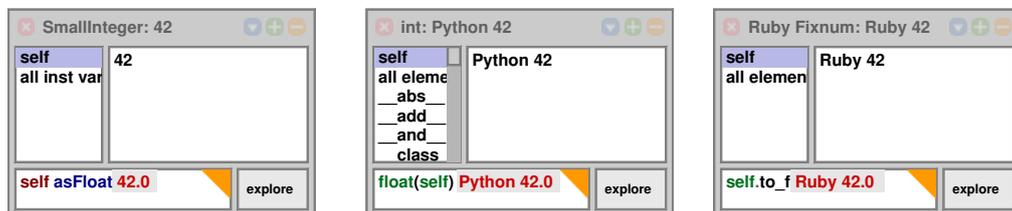
In comparison to that, Figure 5.2a shows SQUIMERA’s workspace in Python mode. This workspace instance is used to evaluate an example of a web mining module called PATTERN [31]. The first line performs Python imports that are needed for the example. Then, a domain-specific Datasheet object is instantiated with different rows and fields and also stored in a variable called `ds`. Finally, the developer again performs a Smalltalk-style *printIt* on `ds` which causes the workspace to display the string representation of the Python object.

As we can see, the interaction with the Python workspace is identical to the original Squeak/Smalltalk workspace: code can be written and modified, as well as evaluated using not only *printIts*, but also *doIts*, *inspectIts*, and *exploreIts*.

When the developer decides to inspect the object stored in `ds`, SQUIMERA’s inspector is opened on the Python object as shown in Figure 5.2b. The window title contains the type of the object as well as its text representation. `self` is again bound to the object, which is why the widget on the right displays the full text representation. Moreover, all attributes of the object are listed similarly to how instance variables are listed in Smalltalk. This includes all special method names used by Python internally, such as `__class__` or `__dict__`. This way, developers can immediately see which attributes are defined on the object. In Python, attributes can either be data attributes which correspond to instance variables, or methods which are functions bound to an instance. When the developer clicks on the `pop` attribute of the object, a bound method is shown as illustrated in Figure 5.2c.

In Figure 5.2c, the developer has already performed a *printIt* which executed the Python expression `self.pop()`. Then, the result of the *printIt* is displayed after the cursor. The text representation in the widget on the right has also been changed accordingly. This shows that the inspector tool also works and behaves on Python objects in the same way it behaves on Smalltalk objects. And since SQUIMERA directly adapts the Squeak/Smalltalk tools, no additional work is needed to, for example, implement the immediate feedback mechanism.

The user experience with regard to the tools remains the same when switching to Ruby. Then, the developer can write and evaluate Ruby code in the workspace and inspect Ruby objects in exactly the same way Smalltalk or Python objects can be inspected. Figure 5.3 demonstrates this with the example of objects of the different languages representing the same number. The syntax to use is determined by the inspected object. Syntax highlighting adjusts automatically.



(a) Interaction with a Smalltalk SmallInteger object

(b) Interaction with a Python int object

(c) Interaction with a Ruby Fixnum object

**Figure 5.3:** Syntax and syntax highlighting are determined by the inspected object

## 5.2 Debugging User Experience

As mentioned in section 2.4, there are many different ways to debug a program. Interactive and live debuggers offer more functionality and flexibility compared to other debugging options, such as commandline-based post-mortem debuggers. In addition to that, the different debugging scenarios described in the following are also great examples of how well language integrations in SQUIMERA work, because debuggers usually have to heavily interact with the underlying execution environment.

### 5.2.1 Debugging Unhandled Exceptions

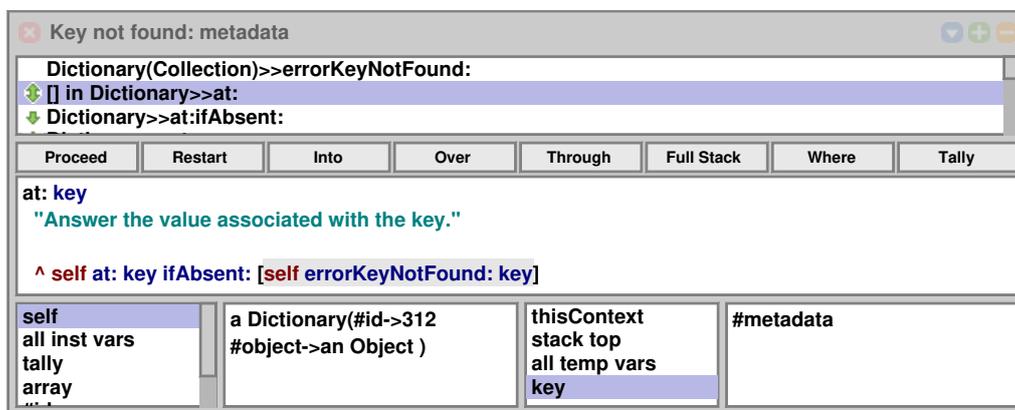


Figure 5.4: Debugging a KeyNotFound exception in Smalltalk

Figure 5.4 shows the current version of Squeak/Smalltalk’s debugger, which still works like the debugger described in section 2.2. This debugger is opened on an unhandled `KeyNotFound` exception. The window title contains the exception name as well as an exception message which state that the `metadata` key was not found. By going through the list of frames, we can find out how this exception was raised in the running program. Further, the two inspector-like widgets on the bottom reveal that the method `Dictionary>>at:` was called on a `Dictionary` with `#metadata` as the key argument. However, this dictionary only contains the keys `#id` and `#object` which is why the exception was thrown. A developer would now be able to, for instance, either use `Dictionary>>at:ifAbsent:` in the program and implement fallback behavior. Or it is possible to interact with the receiver object and add a new value for the missing `#metadata` key. Finally, the developer can control the execution using the debugger’s button bar.

SQUIMERA is able to detect unhandled exceptions in Ruby and Python applications and opens a debugger accordingly. Since the detection of such exceptions is not part of common virtual machines for neither Python nor Ruby, we implemented it as part of the language integration as described in subsection 4.1.2.

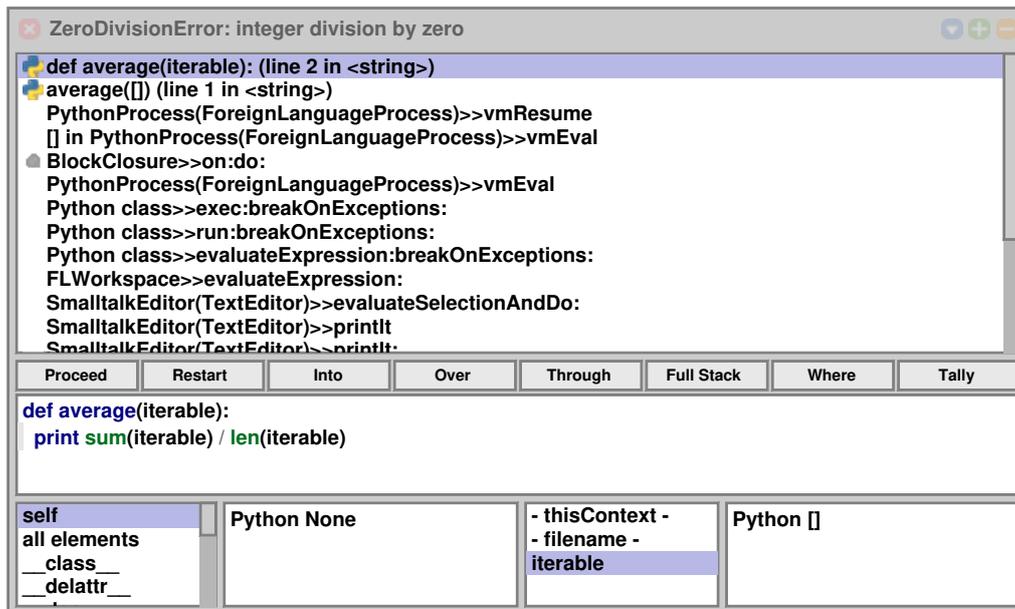


Figure 5.5: Debugging a ZeroDivisionError in Python

Figure 5.5 shows a debugger window presented to the user when an unhandled `ZeroDivisionError` occurs in a Python program. The window's title displays the type of the Python exception as well as the error message associated with it. Moreover, there is a list of stack frames that have led to the exception. The top two frames are Python frames which is indicated by the Python icons in the list. Moreover, the Python expression or function currently being executed is mentioned as well as the corresponding line and filename.

In this example, the exception was thrown during the execution of a Smalltalk `printIt` which is why there is no file for the code. Instead, `<string>` is used which is a Python convention for dynamically compiling code at run-time. Additionally, this is the reason why there are Smalltalk frames on the call stack that, for example, perform the `printIt` method on a `SmalltalkEditor`. The code editor widget of the debugger shows the code being executed in the selected frame with syntax highlighting enabled. The current line of execution is also highlighted, just like in the original Smalltalk debugger. Since the code has been executed in the global Python context, the left half of the bottom lists the Python globals. The right half provides more information on the Python context attached to the frame: `"- thisContext -"` is a reference to the frame object and `"- filename -"` displays the current filename. Then, the local variables are listed. Since there is only the variable `iterable`, the developer is immediately able to identify the root cause of the problem. The naively implemented `average` function does not check if the `iterable` is empty, and in this case a `ZeroDivisionError` is thrown.

At this point, the developer may further inspect stack frames in order to, for example, examine why an empty `iterable` was provided. It is also possible to modify code and then restart a specific frame. Or the developer may decide to



For demonstration purposes, we run OPTCARROT, a NES emulator written in Ruby [37], with SQUIMERA. This is possible, because the emulator is supported by the Topaz interpreter and it uses the same engine for rendering and input handling as RSqueak/VM, namely Simple DirectMedia Layer (SDL) [66]. After a couple of seconds, the emulator opens a new window and starts to draw the current NES ROM. At this stage, we press the interrupt key which triggers a user interrupt. Since the Ruby process is taking up the most computing resources at this moment, it is likely that the user interrupt happens during the execution of the emulator. Then, a minimal debugger window as shown in Figure 5.7 is opened. Its window title indicates that it has been triggered by a user interrupt. The developer then can choose to *Proceed*, *Abandon*, or *Debug* the process on which the debugger was opened. While the window is opened, the process is no longer scheduled for execution. Therefore, the execution of the NES emulator is suspended.

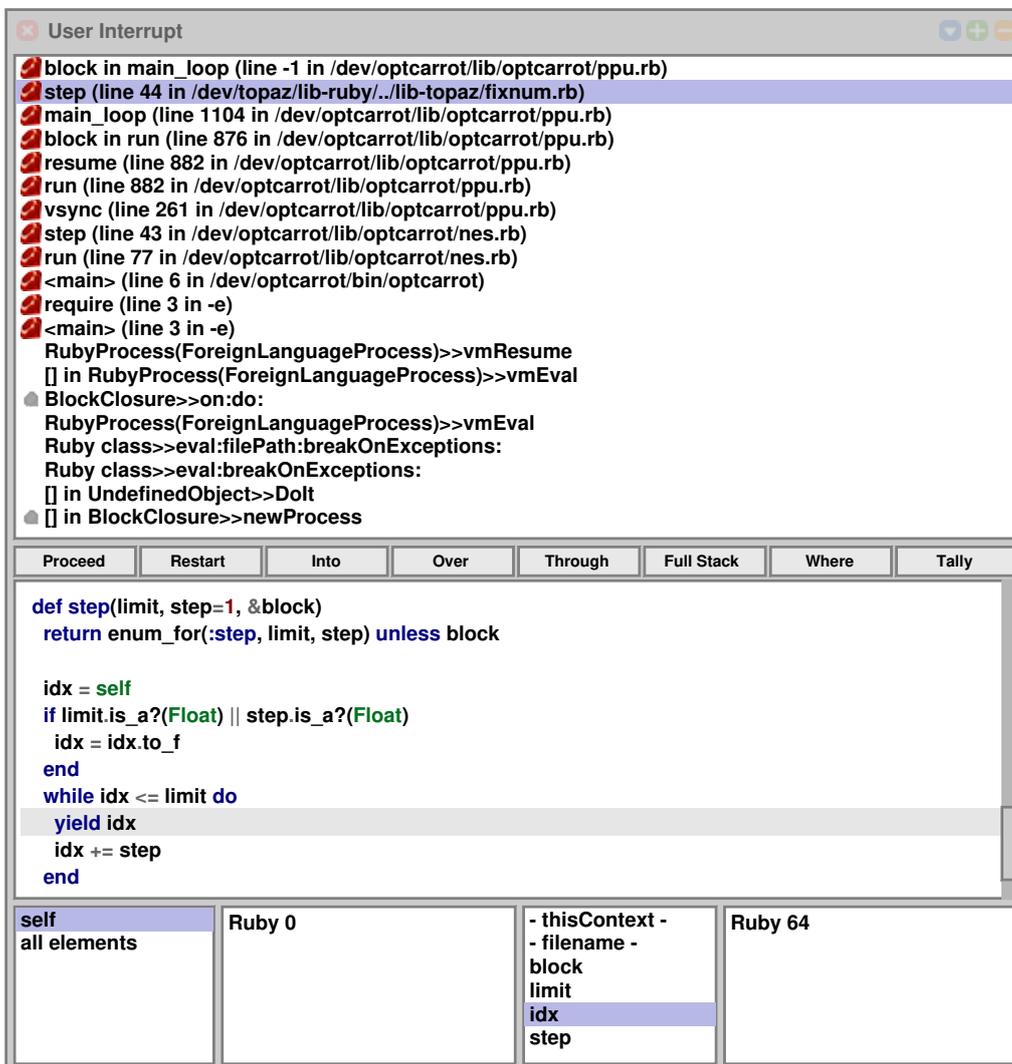


Figure 5.8: Debugging a Ruby process while it is running

Figure 5.8 shows the full debugger window opened after the developer decided to debug the process. Ruby stack frames are highlighted with a Ruby icon, similar to how Python frames were highlighted in Figure 5.5. This time, there are many Ruby stack frames, most of them, including the top frame, execute code from a file called `ppu.rb`, in which the Picture Processing Unit of the NES is implemented. Therefore, the emulator was refreshing its display at the time the interrupt was triggered.

In the example, the second frame is selected which executes code from the `Fixnum` class which is part of Ruby’s standard library. The Ruby code from line 1104 in the `ppu.rb` file that caused a new frame in `fixnum.rb` is `0.step(248, 8) do`. Therefore, the value of `self` in the execution context is `0`, while `limit` is set to 248 and `step` to 8. All this information is accessible with a few clicks in the debugger. Following the stack frames even further, we find the Ruby `<main>` frame. This frame is followed by the first Smalltalk frame which is responsible for resuming the language process. The last but one frame contains the `doIt` that initially started the execution of `OPTCARROT` followed by the bottom frame which created a new Smalltalk-level process for this.

Hence, the debugger can be used to inspect and modify the running emulator including frames executing its own code, but also code from the standard library, as well as `SQUIMERA`’s code that facilitates the execution of the Ruby language process and also Smalltalk code that initiated everything.

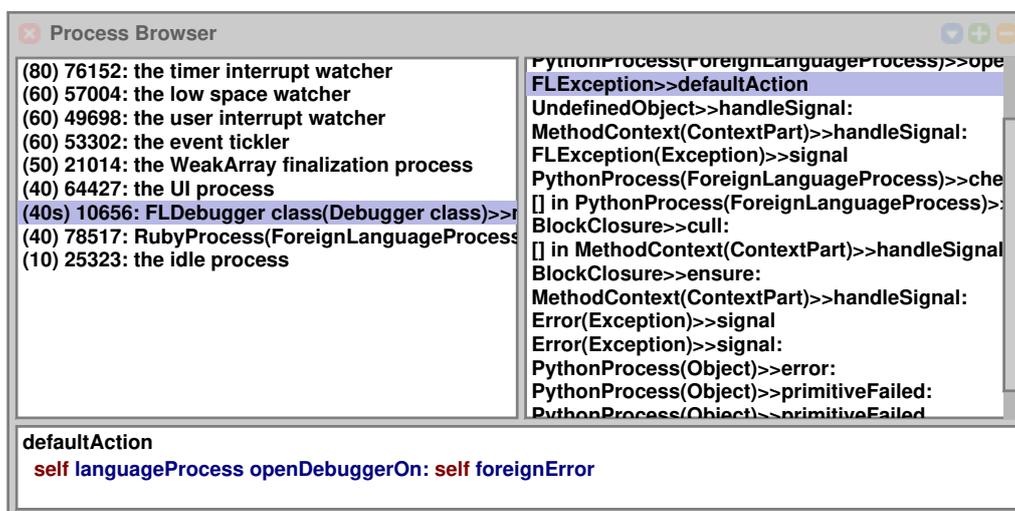


Figure 5.9: Observing language processes in Squeak/Smalltalk’s process browser

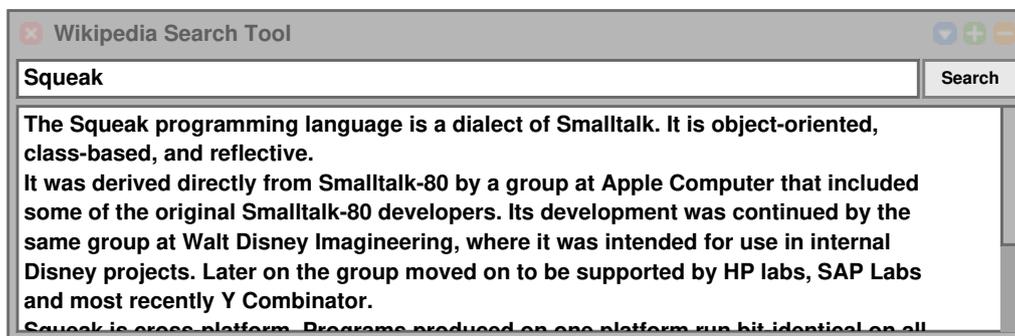
To further understand how language processes work in `SQUIMERA`, we have a look at Squeak/Smalltalk’s process browser as shown in Figure 5.9. The list on the left contains all existing processes, including common Squeak-specific processes such as “the user interrupt watcher” or “the UI process”.

Additionally, two language processes are being executed at the time the screenshot was taken. The selected process is a Python process and the next one in the list is a Ruby process. The latter is running as normal while the former is currently being debugged. This is indicated with an “s” for *suspended* which is added as a suffix to the priority, 40 in this case. Also, this is the reason why the top frame executes a method in `FLDebugger`. The code of the selected stack frame is displayed on the bottom of the process browser. The line of code currently being executed as part of the frame is highlighted in red and suggests that a `FLException` was thrown during the execution of Python code. Therefore, this screenshot shows how `SQUIMERA`’s debugger was opened on an exception in a Python language process.

The Ruby process as well as all other processes are not affected by this. Nonetheless, it is possible to also attach a debugger to them, or to suspend, resume, or terminate them using the context menu on the process list.

### 5.3 Reusing Software Libraries

As discussed in section 1.2, developers build modular software which is crucial to support extensibility and reusability. However, the reusability aspect is often limited by language boundaries, because developers can only reuse software written in their program’s language. In section 3.2, we explained how our approach allows not only reuse of software development tools, but also of software libraries and frameworks in a convenient way.



**Figure 5.10:** A simple search tool for Wikipedia written in Smalltalk which uses a Python library to communicate with the Wikipedia API

To understand why this is beneficial for developers, we have to have a look at how software written in other languages is normally reused, for example in Smalltalk. For this reason, we have built a simple tool, shown in Figure 5.10, that can be used to retrieve summaries for a search term from Wikipedia. To implement this tool, we could have written an algorithm in Smalltalk that communicates with the Wikipedia API [72]. In order to be able to understand how the API can be

used, we would need to read its documentation first. Then we find out that the API returns data in the JSON format, so we would need to install and use a JSON parser in Squeak/Smalltalk. Instead, we chose to use the Python library WIKIPEDIA [50], which already provides a function to directly retrieve a Wikipedia summary for a given search term.

---

**Listing 5.1:** Example method calling out to a Python program via `OSProcess`

---

```

WikipediaToolFFI>>contents
| command result |
self searchTerm ifEmpty: [ ↑ '' ].
command := '/usr/bin/python <1s> "<2s>"' expandMacrosWithArguments: {
  (FileDirectory default / 'wiki.py') fullName. self searchTerm}.
result := (PipeableOSProcess command: command) outputAndError.
↑ result second
ifEmpty: [ result first ]
ifNotEmpty: [ :stderr | (self error: stderr) messageText ]

```

---

Listing 5.1 shows a Smalltalk method that calls out to a Python script “wiki.py” using a `PipeableOSProcess`. This way, “wiki.py” is executed on the default Python interpreter by a process on operating system-level. This script simply takes the last commandline argument as a search term, calls the `summary` function provided by the WIKIPEDIA library, and prints the result to `stdout`. The `stdout` and `stderr` streams are then stored as an array in the temporary variable `result` in Smalltalk. If the error slot is empty, the result from `stdout` is returned. Otherwise, the error message is signaled.

In case of an error, however, one will not be able to debug the Python program from Smalltalk. Instead, `stderr` can return an error string which may help to understand the problem. Otherwise, the Python program needs to be debugged with PDB or in a Python IDE.

---

**Listing 5.2:** Alternative method directly calling `summary` in the WIKIPEDIA library

---

```

WikipediaTool>>contents
↑ ((Python eval: 'wikipedia.summary')
  __call__: self searchTerm) asSmalltalk

```

---

Listing 5.2 shows a drop-in replacement made possible by SQUIMERA for the code in Listing 5.1. The `summary` function is first retrieved from the `wikipedia` module. Then it is called with the search term as the argument and lastly the result is converted to a Smalltalk string and returned.

First, the second version is much more concise. Not only the method has fewer lines, but also the “`wiki.py`” script is not needed anymore. This means responsibility is not split and it is not necessary to use the indirection via the operating system. More importantly, however, a debugger will be opened in case of errors thrown in the `WIKIPEDIA` library. This debugger can be used to understand the interaction between the two languages as well as potential problems in the library.

Similarly, it is possible to reuse other Python or Ruby libraries including their standard libraries. This can not only be used to reuse libraries and tools for specific purposes, but also to compensate shortcomings of the Squeak/Smalltalk standard library. Ruby’s regular expression engine is, for instance, superior in many scenarios to the regular expression matcher that ships with Squeak. As mentioned in subsection 4.2.4, SQUIMERA’S tools use the Python library `PYGMENTS` for syntax highlighting.

## 5.4 Limitations of Squimera

In this section, we discuss the limitations of the SQUIMERA system in more detail. Due to time constraints, we were unable to fully implement the ability to call Smalltalk methods from foreign languages. As discussed in section 3.2, this would allow us to reuse software frameworks, which is currently not entirely possible with SQUIMERA.

We demonstrate this with another example, which also allows us to discuss further limitations of SQUIMERA. This time, we intend to reuse `FLASK`, a web server framework written in Python [54]. Listing 5.3 and Listing 5.4 show a Python and a Smalltalk code snippet that can be used to start a `FLASK` server in SQUIMERA. First, the Python code needs to be evaluated in a Python workspace. This globally defines a variable `server_data` as well as a Python function `start_server`. Afterwards, it is possible to call this function from Smalltalk with the second snippet in a Smalltalk workspace. As a result, the framework prints a local URL to `stdout` that can be opened in a web browser. The web page displays a list of all class categories available in the Squeak/Smalltalk environment. Each category is linked to a page that in turn simply lists all Smalltalk classes of the selected category.

Even though we are able to start a server as part of a software framework, we are restricted in the way we can do it. If we, for instance, add a new category or class, the web server is not automatically updated, as it can only call Python functions which retrieve their data from the `server_data` variable. The variable is globally available, so we could, for example, update it frequently from Smalltalk to ensure the web server serves the latest lists.

**Listing 5.3:** A simple FLASK server example

---

```

server_data = {}

def start_server(new_server_data):
    global server_data
    from flask import Flask, render_template_string
    app = Flask(__name__)
    server_data = dict(new_server_data)

    @app.route('/')
    def index():
        return render_template_string("""
            <ul>
            {% for c in categories %}
                <li><a href="/{{c}}">{{c}}</a></li>
            {% endfor %}
            </ul>
            """, categories=server_data.keys())

    @app.route('/<category>')
    def list_classes(category):
        return render_template_string("""
            <ul>
            {% for class in classes %}
                <li>{{class}}</li>
            {% endfor %}
            </ul>
            """, classes=server_data.get(category, []))

app.run()

```

---

**Listing 5.4:** A Smalltalk *dolt* that starts the FLASK example as shown in Listing 5.3 providing a Smalltalk Array of Arrays as data argument

---

```

(Python eval: 'start_server') __call__: (
  SystemOrganizer default categories collect: [:each |
    {each asString.
      (SystemOrganizer default listAtCategoryNamed: each)
        collect: [:ea | ea asString]})

```

---

This, however, is usually not how frameworks are used. Instead, the callbacks should be able to retrieve data directly from Smalltalk, which would ensure that up-to-date data is always being served.

Nevertheless, there is another problem that this example reveals. `FLASK` uses Python's `SocketServer` to bind the server to a local socket. This socket, however, is blocking, which means that the Python interpreter blocks as long as there are no incoming requests. Consequently, the interpreter cannot yield back to Smalltalk. And since the Smalltalk scheduler cannot switch to the UI process anymore, we are unable to interact with the Squeak/Smalltalk environment. After a number of requests to the server, the Smalltalk environment is being updated. Having to send requests to the server, however, should not be a requirement to be able to work with the environment. Therefore, blocking sockets are currently a problem in `SQUIMERA`.

Additionally, `FLASK` catches all exceptions thrown in routing functions to ensure that the server does not crash. Therefore, the `SQUIMERA` debugger will not be opened, because of the framework. This, however, is a common pattern used in frameworks. The ability to selectively open a debugger on special user-defined exceptions would solve this problem in `SQUIMERA`, but is currently not supported.

## 5.5 Further Limitations of Our Approach

In addition to the shortcomings with regard to `SQUIMERA`'s current implementation, there are further limitations of our approach that we now discuss in more detail.

**Conceptual Mismatches of Languages and Tools** First of all, we have only integrated object-oriented programming languages as part of `SQUIMERA`. One reason for this is that these languages are relatively similar to Smalltalk. This makes it, for instance, easier to map Smalltalk semantics to the different languages and to conceptually reuse the Smalltalk tools. Nonetheless, we do not believe that it is impractical to integrate languages following other programming paradigms. But this might require additional work.

However, we noticed that there are also some conceptual problems with regard to the Smalltalk tools with our integration of Python and Ruby. These may also occur when integrating other languages. Squeak/Smalltalk's system browser tool, for example, manages all source code in objects, because everything in Smalltalk is an object. But this means that the objects are part of the environment and saved in Smalltalk image files. Just like many other languages, source code for Python and Ruby is managed in files on the file system. This in turn implies that an adapted version of the system browser would need to operate on files rather than on objects.

In addition to that, the system browser enforces a certain structure consisting of class categories, classes, method protocols, and methods. However, it may not be reasonable to enforce the same structure on Ruby or Python applications. Instead,

a tool similar to a conventional code editor that uses the file system might be a better fit for these languages.

We, however, experimented with adapting the system browser for Python and are able to add new classes and methods accordingly. After all, we encountered different synchronization problems, because we decided to also manage Python in Smalltalk objects to keep the number of changes to the browser tool as low as possible. Methods, for instance, can also be added to Python classes using metaprogramming. But this does not update the objects holding the method source code in the Smalltalk world. Even though these methods exist, they are not listed in the adapted browser. Again, this requires to be able to call Smalltalk from Python.

According to Cunningham [30], one lesson learned from VisualAge for Java is that only people with Smalltalk experience enjoy the tools. One important reason for that is that the tools are written in Smalltalk, so Smalltalk experience is required to change them. The ability to call Smalltalk from Python might yet again help with this problem. When this is possible, we could implement a Python library that wraps around Squeak/Smalltalk's `ToolBuilder API`. This would allow us and other developers to build and modify tools for the Smalltalk environment entirely in Python.

Furthermore, the ability to call Smalltalk from foreign languages also has a downside. Currently, Ruby and Python methods can be called with Smalltalk objects as arguments. The `VM` then converts them to corresponding objects of the target language which currently happens automatically, but only for primitive data types. Therefore, once the other way is also possible, it is necessary to also provide converting methods for each integrated language, for example `asRuby` or `asPython`. Then, however, the automatic conversion needs to be turned off which would give developers more control about the objects being passed to each language. On the other hand, this would also result in less concise code and additional work, as developers always need to think about the type conversion from one to another language as well.

**Detection of Unhandled Exceptions** As part of section 3.1, we explained that `SQUIMERA` needs to be able to detect unhandled exceptions as soon as they are thrown to avoid stack unwinding. Some languages, including Python and Ruby, however, use the termination model of exception handling. It is comparably easy to implement and yet powerful during the execution of a language. Nonetheless, it makes it quite hard to provide useful debugging facilities.

To avoid unwinding the call stack, `SQUIMERA` currently performs bytecode analysis. First, it works on a different abstraction layer than the actual exception handling. That is because the language specification does not consider unhandled exceptions the same way they are considered in Smalltalk for example. It is not a requirement for implementing the termination model of exception handling.

The bytecode analysis, on the other hand, is also error prone and it cannot reliably detect unhandled exceptions in all cases. The reason for this is that there are too many different ways to define try-except statements or to mask exceptions

with builtins in Python. Python is too dynamic in that sense. The same holds true for Ruby.

Instead of performing bytecode analysis, SQUIMERA could analyze the source code. This again would be on a different abstraction level. Moreover, this requires to be able to have access to all code in the system, including the code that may have been dynamically added through the use of metaprogramming. Nonetheless, performing an analysis on, for example, the AST of the code in question raises similar problems compared with bytecode analysis. For instance, if functions are used to dynamically define exceptions to catch in a try-except statement, these functions need to be evaluated somehow. In the worst case, the functions have side effects that manipulate the exception handling even further.

Another approach to detect unhandled exception would be to fork the program somehow whenever an exception is raised and check whether the exception is propagated to the main stack frame. This way, it would be possible to reliably detect unhandled exceptions. We, however, rejected this approach, because it is impractical to manage all these forks. But more importantly, this would make the problem of side effects even worse. Developers just do not expect parts of their program running in multiple processes or coroutines at the same time.

**Squimera’s Performance** There are different aspects in SQUIMERA that can affect performance. The main impact performance-wise is introduced by using Smalltalk-level processes to execute interpreters of foreign languages. This directly impacts the number of bytecodes processed by each interpreter loop significantly.

In case of our Python integration, however, a ballpark measurement of the Richards benchmark [90] shows that Python programs perform similar to CPython in SQUIMERA, even though we use the PyPy interpreter. Therefore, one can say that Smalltalk-level processes decrease the performance almost as much as performance can be increased by PyPy’s JIT compiler. On the other hand, a full-fledged IDE is now running at the same time and in the same operating system process. The performance overhead can be controlled with the `sys.setcheckinterval()` function in Python. The default check interval value is 10,000. Increasing the value, increases the performance of Python programs, but lowers the responsiveness of the SQUIMERA environment. Decreasing it causes Python processes to yield back to Smalltalk earlier.

As demonstrated in subsection 5.2.2, we are able to run the NES emulator OPTCARROT on top of SQUIMERA. The emulator is usually able to reach a little more than 28 frames per second on Topaz. On SQUIMERA it only reaches around one frame per second when the check interval is set to 10,000. In our Ruby integration, the value of the check interval can currently be changed with an environment variable. Disabling the process switching completely, lets OPTCARROT run at 28 frames per second again.

Therefore, the overhead introduced by our unhandled exception detection is rather insignificant. This overhead depends mainly on the use of exception handling in a program. Nonetheless, the frame that handles an exception is usually not far away from the frame that has thrown the exception. Additionally, the total number

of frames is usually rather low. In case of an unhandled exception, all frames have to be traversed. This overhead, however, can be considered irrelevant, because performance is not as important during debugging.

## 5.6 Summary

We think that the tool adaptations of SQUIMERA feel and behave consistently across all integrated languages which improves the programming experience for developers. As part of that, SQUIMERA supports live object exploration as well as various debugging scenarios and provides means to reuse and combine software written in different languages. Some limitations remain, for example with regard to the detection of unhandled exceptions or programs that use blocking sockets.

## 6 Related Work

In this chapter, we discuss solutions and technologies related to our work. Further, we compare them with our approach and with the SQUIMERA system.

### 6.1 Tools and Integrated Development Environments

First, we describe different and popular tools and IDEs and compare them to the SQUIMERA programming environment.

#### 6.1.1 Development Tools for Python and Ruby

To understand what kind of problems SQUIMERA solves with regard to Python and Ruby development, we have to have a closer look at the current development tools for each language.

**Writing Code in Ruby or Python** First of all, Ruby and Python are, just like many other programming languages, file-based. Therefore, many developers use extendable code editors like Emacs [100], Sublime Text [97], or Atom [46] to write their programs.

Even though these editors can be extended with useful programming features, they usually do not integrate with the runtime and hence cannot support features such as debugging in the same way IDEs can.

Furthermore, both languages provide interactive Read–eval–print Loops (REPLs) that can be used similarly to SQUIMERA’s workspace. However, developers can only evaluate code and see the result which is comparable to Smalltalk *printIts*. This makes it rather inconvenient to inspect certain objects, as it usually takes more effort to drill down on the right aspects of an objects in a REPL than by using tools like the Smalltalk inspector or object explorer. More importantly, the Smalltalk tools can provide live feedback while it is only possible to manually poll for new data in a REPL. In addition, an interactive shell is not as visual as an interactive programming environment.

**Python Debugger PDB** Python’s standard library contains a debugger called PDB [85]. This module can be invoked to debug other Python scripts which enables post-mortem debugging as soon as a program fails unexpectedly. Moreover, developers can add PDB-specific code to their programs to set breakpoints. The debugging sessions are provided through an interactive shell that can be used to

inspect the state of the program as well as to modify, for example, objects and variables.

PDB is completely implemented in Python and does not integrate with special components of the interpreter. Unlike in Smalltalk, Python features like exception handling are part of the run-time. As a result of that, PDB is limited in functionality compared with SQUIMERA, as it cannot provide edit-and-continue debugging for example. While SQUIMERA allows restart of specific frames with updated code, PDB can only restart the entire program.

**Debugging Tools for Ruby** Ruby comes with a debugging library called `DEBUGGER__` [93] which works similar to PDB in Python. An interactive shell can be used to inspect run-time exceptions. Similarly, the `DEBUGGER__` library is limited in terms of debugging capabilities, while SQUIMERA also supports edit-and-continue debugging for Ruby.

Additionally, some Ruby developers also use `PRY`, an alternative Ruby shell [69]. This shell enhances the programming experience with very specialized features for Ruby including commands that are Smalltalk-inspired. It also can be used as a debugger to some extent. Its capabilities are nonetheless also limited with regard to, for example, fix-and-continue debugging.

The different tools for Ruby and Python are specifically designed for each language. Therefore, the programming experience differs between the two, although Python and Ruby are relatively similar programming languages. The mismatch between the behavior of the tools hence implies additional work for developers when learning to develop in each language.

In contrast, the programming experience in SQUIMERA is consistent, because the same set of tools can be used for all supported languages. The debugger, for instance, always feels and behaves the same, no matter which language is debugged. The same holds true for all other tool adaptations.

### 6.1.2 Integrated Development Environments for Python and Ruby

There are various IDEs that provide a great number of development tools for Python and Ruby.

**PyCharm and RubyMine** PyCharm [60] and RubyMine [61] are customizable and cross-platform IDEs for Python and Ruby developed by JetBrains. Both come with intelligent code editors, refactoring and inspection tools, builtin support for version control systems, deployment, and remote development, as well as tools for debugging, profiling, and testing. Moreover, PyCharm offers support for popular Python web frameworks like Django and RubyMine for Ruby on Rails, a popular web framework written in Ruby. Both IDEs can also be used to develop HTML, JavaScript, and other languages for the web. Furthermore, it is not only possible to use different Python or Ruby interpreters locally in these IDEs, but also in a remote setup.

**Wing Python IDE** The Wing Python IDE [116] by Wingware is similar to PyCharm. It also has an intelligent code editor and similar features with regard to debugging, object inspection, testing, remote development, and refactoring. Furthermore, its professional version also comes with support for different web frameworks for Python.

**Eclipse, PyDev, and Aptana Studio** Eclipse [32] is an IDE framework with support for many programming languages. It was initially built for Java development and still is a popular IDE among Java developers. It is supported by a large community that has built many plugins for Eclipse. Not only can these plugins extend Eclipse with more tools for Java, but there are also plugins that allow adaptation of Eclipse's tools in some ways for other languages.

The Dynamic Languages Toolkit (DLTK) [105], for example, uses Eclipse's support for plugins and aims to help to adapt its tools for dynamic languages. It also comes with exemplary development environments for Ruby and Python.

PyDev [20], on the other hand, is a plugin developed by Appcelerator which integrates only Python into Eclipse. It provides very similar features to the other Python IDEs including code completion, refactoring and testing tools, Django integration, and various debugging features.

Aptana Studio [2] is based on Eclipse and is mainly an IDE for web development. As part of that, it not only supports web languages like JavaScript, HTML, or CSS, but also many other languages that are used to implement, for example, web servers. Therefore, it provides tools for Python development through the PyDev plugin, but is also popular among Ruby developers, because it integrates Ruby on Rails similar to RubyMine.

**Visual Studio** Visual Studio [75] is an IDE from Microsoft which also supports multiple programming languages. According to the Top IDE index as of May 2017 [24], Visual Studio recently surpassed Eclipse as the most popular general purpose IDE. It has builtin support for almost all languages that are used for Microsoft's products including C, C++, C#, TypeScript, and VB.NET. Over the years, it has been extended with support for many other languages such as Python, Ruby, or JavaScript.

Compared with our prototype system SQUIMERA, all of these IDEs provide a lot more and mature features for Ruby and Python development, as they have been in active development by larger communities and companies for years, in some cases even decades. As mentioned in chapter 3, however, all of these IDEs are based on the architecture illustrated in Figure 3.1a. Even though most of them support different Ruby and Python interpreters, their debugging capabilities are in some ways limited compared with SQUIMERA.

On the other hand, the SQUIMERA programming environment runs in the same process that also executes the languages, following the Smalltalk architecture as shown in Figure 3.1b. This gives more control over the execution of different languages, as it supports to restart frames and patch code at run-time for example.

This in turn enables edit-and-continue debugging in SQUIMERA, while all other IDEs are limited as to how they can modify a program while it is running.

Additionally, most of these IDEs use a different approach to SQUIMERA when it comes to detecting unhandled exceptions for debugging purposes. PyCharm's debugger, for instance, supports conditional breakpoints on user-defined exceptions as well as different activation policies. Its unhandled exception detection is based on source code analysis which is performed as soon as exceptions are raised. Further, it is possible to ignore certain library files in PyCharm, because its detection also does not always work reliably. A similar approach is applied in the Wing IDE, Visual Studio, and others. Their debuggers have many configuration options that help to drill down on certain exceptions. This helps to minimize cases in which a debugger is expected, but not opened, and vice-versa.

In contrast, SQUIMERA performs bytecode analysis which leads to very similar problems with regard to the detection of unhandled exceptions. Furthermore, it would make sense to add similar configuration options to the debugger, because the detection cannot work reliably as explained in section 5.5.

As part of Visual Studio's support for Python development, it is possible to expose C or C++ interfaces via Python's C API, so that C or C++ code can be used to, for example, speed up Python applications. Additionally, Visual Studio supports mixed-mode debugging. With this, developers can debug Python and native C or C++ at the same time, with combined call stacks.

SQUIMERA also allows developers to debug Smalltalk and Python code, as well as Ruby code. Currently, Ruby and Python frames always show up on top of a call stack in SQUIMERA's debugger. As soon as Python and Ruby can call back to Smalltalk, it is possible to observe mixed call stacks with more than two languages and in any order. Only the bottom frames will remain Smalltalk frames, because Smalltalk acts as the host for all integrated programming languages.

Moreover, there are different IDE extensions, such as the LIVE-PY-PLUGIN [63] for PyDev and Eclipse, that enable live coding capabilities to some extent for various dynamic languages. However, these systems usually reload programs entirely and therefore cannot provide a higher level of liveness than Tanimoto's level 3 [104]. On the other hand, live programming environments like Squeak/Smalltalk, which SQUIMERA is based on, allow modification of programs at run-time and are able to provide visual feedback in a few hundreds of milliseconds [86].

Further, some IDEs like the Wing Python IDE, RubyMine, or PyCharm are specialized for specific languages. Some of them even integrate with popular frameworks for each language. This however means that developers need to read corresponding documentations and do tutorials as well as additional trainings in order to be able to use all features of each IDE. The downside is that this knowledge often only helps to develop in one specific environment, and only few aspects are useful when developing in other programming languages and in other IDEs.

General purpose IDEs or IDEs with support for multiple languages such as Eclipse or Visual Studio often have other disadvantages. In Eclipse, for instance, many tools work best with Java, the language Eclipse was designed for originally. One example for that is its debugger. It communicates with the Java Virtual Machine

Tools Interface (JVMTI) to control Java processes. Although the runtime supports hot-swapping code for method bodies in Java, this feature is not exposed in the JVMTI.

For one, this means that developers are able to modify their Java programs at run-time as long as they modify existing methods. In almost all other scenarios, a program needs to be restarted which may be confusing for beginners.

On the other hand, it is unclear why hot-swapping is not part of the JVMTI which would potentially encourage other integrations to also allow this feature. Nonetheless, DLTK or PyDev, as previously explained, cannot provide this feature, because hot-swapping is not supported by common Python or Ruby run-times. Hence, the development experience changes with the language in Eclipse.

In SQUIMERA, on the other side, all tools behave consistently across all languages. This means that developers can reuse their knowledge of the environment for all languages they develop in, as they only have to get used to one IDE.

### 6.1.3 Other Related Tools for Software Development

In addition to the previously discussed development tools for Ruby and Python, there are other tools and IDEs related to our approach and SQUIMERA.

**IBM VisualAge** IBM VisualAge was a family of mainly Smalltalk-based IDEs with support for different programming languages. In VisualAge for Java, a Smalltalk VM was used with support for both, Smalltalk and Java [25].

This demonstrated that it is possible to use a Smalltalk environment for tools as well as a multi-language VM for the execution of another language in a commercial context. Instead of being a family of IDEs, SQUIMERA is a single IDE with support for multiple languages and with consistent tooling.

**Helvetia** Renggli et al. have worked on embedding DSLs into Smalltalk, so that the tools provided by the environment can also be used for these languages [87]. In order to do so, their Helvetia system transforms code written in a DSL to Smalltalk code on which the tools can operate.

In SQUIMERA, on the other hand, each programming language is executed by a corresponding interpreter. More importantly, to be able to work with integrated languages in SQUIMERA, we have adapted Squeak/Smalltalk tools instead.

**Smalltalk/X** STX:LIBJAVA [56] and SmallRuby [113] are experimental Java and Ruby implementations on top of Smalltalk/X. The former project uses a VM with additional support for Java bytecode instructions, while the latter compiles Ruby source code to Smalltalk/X bytecode which is then executed by the VM. In addition, both projects allow interoperability with Smalltalk.

Similar to SQUIMERA, SmallRuby and STX:LIBJAVA also provide Smalltalk-based development tools for Ruby and Java. Objects of these two languages expose the same meta-object protocol as normal Smalltalk objects which ensures that the original tools can operate on them. This way, the Smalltalk/X debugger, for exam-

ple, also supports mixed stacks. SQUIMERA also aligns the meta-object protocol for foreign objects in a similar way. Additionally, it removes unneeded and sometimes hindering functionality specific to Squeak/Smalltalk by adapting the tools, and provides abstract interfaces which can be used to add more languages. This further reduces the work that needs to be done to integrate additional languages.

**Multi-language Debugger Architecture** Vraný and Piše proposed an architecture that combines different debuggers into one debugger with multi-language support [114]. For this, they suggest a generic debugging interface as an abstraction layer which is then used by a multi-language debugger.

As part of our approach, we presented an interface on the level of the execution environment instead. This not only allows developers to use only one debugger in a similar way, but also ensures that the different interpreters support the same capabilities. This in turn is a requirement to be able to provide a consistent debugging experience with support for edit-and-continue debugging. In contrast, their architecture only allows the intersection of all debugging features provided for all languages.

**Eco** Diekmann and Tratt presented an approach for syntax-directed-style editors [34]. Their prototype editor Eco allows developers to write composed programs and supports Python, HTML, and SQL. Internally, the editor uses language boxes to allow nesting of code written in the different languages. Eco then parses the different code snippets with a custom parser and translates a composed program into a Python script. HTML elements are translated to Python print statements, while SQL code is translated to corresponding SQL API calls in Python.

Although SQUIMERA also allows developers to mix languages in a program, it works quite different compared with Eco. Instead of translating source code written in different languages to one common language, each source code is executed by a corresponding interpreter. However, an Eco-like editing experience could also be possible with SQUIMERA. Instead of using different parsers for each language box, the code needs to be evaluated by the “eval” primitive of the corresponding language plugin. This, however, would require that each child language box returns objects of the language of its parent box, but this is something the tooling or the VM can help with.

## 6.2 Cross-language Integration Techniques

The idea to use more than one programming language in software programs has existed for decades. In the late 1960s, Cheatham identified three approaches that give developers more diversity in terms of language features [26]. The first approach is to come up with a variety of programming languages for different use cases. The second suggests to use a “shell” language providing a universal set of features that can then be used across all languages. Lastly, he mentions the idea of

an extensible language facility. This facility includes a core language and means to extend it for different use cases.

Today, we have a large number of different programming languages. Hence, Cheatham’s approach 1 has been applied even though it comes with many drawbacks. Further, one could argue that approach number two has led, for example, to the notion of foreign function interfaces and the use of inter-process communication, ultimately using the operating system or a network connection as the abstraction layer between languages. As a result of the third approach, many extensible languages have evolved which support syntactic and semantic modifications [121].

Our approach, on the other hand, acknowledges the fact that there are many different languages and attempts to make them interoperable and more useful. For this, we are using Smalltalk as a more sophisticated “shell” language with support for high-level language features such as message passing and inheritance.

### 6.2.1 Conventional Language Integration Mechanisms

**Foreign Function Interfaces** With foreign function interfaces, it is possible to call out from a program written in one programming language to programs written in other languages. They are used by developers, not only to be able reuse more software. FFIs can also be used to speed up performance-critical computations that can be implemented more efficiently in non-dynamic languages. Nonetheless, most FFIs provided by popular languages are often based on `libffi` [51] and only support to call lower-level languages like C, as they use the underlying operating system as the abstraction layer.

Python, for instance, comes with a `ctypes` library since version 2.5 which is based on `libffi` and can be used to call functions in shared libraries [84]. Similar to this, Ruby provides the `Fiddle` module [94]. Java supports foreign function calls through the Java Native Interface [67], while a Squeak/Smalltalk VM usually provides the functionality via a special `SqueakFFIPrims` plugin which is also based on `libffi`. Further, wrapper libraries like SWIG [9] have been developed to make it more convenient to work with FFI calls in general.

**Inter-process Communication** Inter-process communication can also be used to call programs written in another language. Compared with the FFI approach, applications using inter-process communication can coexist and communicate with each other, and are therefore less coupled. A prominent example for this are applications that use a database. For that, they usually communicate over a network connection with a database server. The communication is often limited to text and usually does not support the exchange of, for example, language objects. Other examples are microservices that use remote procedure calls to communicate with other services [83].

However, inter-process communication not only introduces performance overheads. Also, developers can often only debug each component separately, because it is rather hard and sometimes impractical to debug multiple processes at once, especially in distributed systems.

As demonstrated in section 5.3, SQUIMERA is able to provide a mechanism for software reuse which we believe is superior in terms of usability to approaches based on FFIS and RPCs. Instead of having to run a subroutine in a separate operating system process, it can be executed in the process of the main program. This allows better control over the execution and ensures that SQUIMERA's debugger remains operational, also for exceptions thrown in subroutines written in different languages.

### 6.2.2 Language Interoperability

In the last decades, a lot of work and research has been done to provide means that allow direct interaction between multiple languages without using an indirection such as the operating system or a network connection.

In 1998, for example, Cleary et al. presented an idea which enables language interoperability for high-performance scientific applications through an Interface Definition Language [27].

Similar to that, Hamilton described how the Common Language Runtime (CLR), which provides the infrastructure for Microsoft's .NET Framework, enables the integration of programming languages on run-time level [55]. For this, the CLR is platform and language neutral and can execute the Common Intermediate Language (CIL) to which all compatible languages have to be compiled to. As a result, CLR supports different programming languages including C#, Java, C++, and Python.

Further and based on the Microsoft .NET Framework, Eaddy and Feiner have presented an edit-and-continue implementation that allows runtime updates to programs running on CLR [36]. However, Visual Studio currently only supports the edit-and-continue feature for C++, C#, and Visual Basic [74], even though it should also work for IronPython according to the paper.

Nonetheless, the CLR approach requires language implementations that can compile a language into an intermediate language. SQUIMERA, on the other hand, makes use of different language implementations written in the same language implementation framework, and runs a different interpreter for each supported language.

**Languages Implemented on Top of Other High-level Languages** In addition, there are various language implementations built on top of other languages. For this, they usually perform some kind of source code transformations which often introduce additional performance overhead.

However, some execution environments perform optimizations, for example JIT compilation, which are able to compensate this overhead, and in some cases even outperform the default execution environment.

JRuby, for instance, is a Ruby implementation on top of Java [80] which is able to outperform Ruby MRI, because it can leverage the JIT compiler of the JVM. Similar to that, MagLev is a Ruby implementation that runs on top of a Smalltalk VM for GemStone/S [99].

In JRuby, Java bytecodes are emitted while MagLev generates bytecodes for the Smalltalk VM. On the other hand, SQUIMERA does not translate programs written in different languages into one bytecode set, but uses different interpreters for each language instead. Also, it uses a tracing JIT compiler to increase the overall performance of all languages.

**Interpreter Composition in RPython** More recent research targets language implementation frameworks, such as RPython and Truffle, that include advanced JIT compiler techniques.

With Unipycation [6], Barrett et al. presented a composition of interpreters for Python and Prolog in RPython, based on PyPy and Pyrolog [16]. Although their bi-language VM is built similarly to SQUIMERA’s VM, it does not have any further scheduling mechanisms for switching between interpreter loops. Also, their work mainly focuses on performance advantages that a meta-tracing JIT compiler can provide rather than on the programming experience or on tooling.

Moreover, Barrett et al. compared Unipycation with other approaches to interpreter composition [5]. For this, they composed each, CPython and PyPy, with a Prolog implementation written in C by using CFFI. Additionally, they also composed a Python and a Prolog implementation on top of Java. As a result, they concluded that the approach used to build Unipycation not only led to a well-performing VM, but was also comparatively easy to implement.

For similar reasons, we used interpreters written in RPython to build SQUIMERA. Instead of having to write language implementations from scratch, or compose interpreters in C, we were able to reuse existing language implementations with relatively low effort, because they are written in the same framework and in a high-level language.

**Truffle’s Polyglot Engine** Furthermore, the Truffle framework in combination with the GraalVM also provides high-performance language interoperability capabilities [52, 117]. The different language implementations, including TruffleJS, TruffleC, and TruffleRuby, emit AST nodes that Truffle can execute and optimize.

Additionally, the framework supports to mix these AST nodes from different languages at run-time as part of its Polyglot engine which enables language interoperability. This way, TruffleC was able to provide support for C extensions in TruffleRuby [53] for example, but is now replaced with Sulong which is able to execute more LLVM-based languages, such as C, C++, and Fortan, on the GraalVM [91].

Although some work has been done to provide debugging support for languages implemented in Truffle [95, 108], the framework focuses on performance rather than on tooling. In contrast, our approach can be seen as a tool-first approach, as we have put more effort into providing a good programming experience than into optimizing the performance of composed languages.

With a Smalltalk implementation in Truffle, we believe it could be possible to leverage its Polyglot engine for multi-language support and tooling similar to SQUIMERA. However, it is unclear to what extent the framework is able to create a

consistent programming experience, as an important part of our approach is the ability to retrofit advanced runtime capabilities such as frame restarting and hot code patching.

## 6.3 Summary

Unlike common `IDES`, the `SQUIMERA` programming environment including all tools run in the same process in which the different languages are executed. This in turn enables, for example, edit-and-continue debugging for languages that usually do not support it. On the other hand, common techniques to integrate and reuse software written in different languages are often hard to debug, as they usually break the tooling. In contrast, different languages can be used in a more concise way in `SQUIMERA` and without breaking, for instance, the debugging experience.

## 7 Conclusion and Future Work

In this report, we have described two common problems developers encounter when working on software. First, they often need to use specific tools for the language they develop in. This requires them to acquire special knowledge which, in some cases, is language-specific and cannot be applied when developing in other languages. We argue that this is one reason developers hesitate to choose a language they are not familiar with, even if that language would be more suitable for their use cases. This problem scales in a business context, where several developers work in teams on a complex software product. Switching to a different language not only requires to migrate code to that language. More importantly, developers need to be trained to be able to work efficiently with the development tools for that language. This, however, can be a substantial financial investment for companies.

On the other hand, software development tools are often built from scratch for each programming language. These tools, however, usually provide very similar functionality, because there are many software development principles that are generally applicable. Code editors often support the same set of features such as code completion, syntax highlighting, and code formatting. Debugging tools usually support the inspection of program state at run-time and can be used to control and manipulate the execution.

The second problem we discussed is the fact that not only tools, but also software libraries and frameworks are often recreated across programming languages. One reason for this is that software is usually built in a modular way which allows for better extensibility as well as reusability. Another reason is that concepts, ideas, or even entire architectures that have proven to work well in one language may also work well in others.

In this work, we presented an approach which attempts to solve both of these problems through reuse. Reusing tools for software development not only reduces the work for language or tool implementers. More importantly, it also makes it easier for developers, as they can work with familiar tools to develop software in different languages in a uniform way. As a result of that, software libraries can also be reused in a more convenient way compared to traditional approaches based on foreign function interfaces for example.

For our approach, we decided to reuse the tools of a Smalltalk environment. In order to be able to unify the programming experience across all languages, we suggest to integrate the different languages not only on IDE-level, but also on language implementation-level. This makes it possible to retrofit powerful runtime capabilities such as frame restarting or hot-swapping, which in turn enable features like edit-and-continue debugging. Language implementation frameworks like RPython make this kind of integration easier.

To demonstrate the advantages of our approach, we have implemented *SQUIMERA*, a Smalltalk-based IDE for Python and Ruby. The *SQUIMERA* VM consists of multiple interpreters and the execution of Ruby and Python programs is coordinated by the Smalltalk scheduler in Smalltalk-level processes. This ensures that the development environment remains responsive, because unlike in conventional IDEs, the user interface as well as Ruby and Python programs are executed in the same operating system process in *SQUIMERA*. By analyzing different use cases, we demonstrated that *SQUIMERA* is able to provide a consistent programming experience across Smalltalk, Python, and Ruby.

There are multiple avenues for future work. As mentioned in section 5.5, it would be useful if foreign languages could call to Smalltalk. This would not only allow the proper reuse of software frameworks in Smalltalk, but also reuse of Python libraries and frameworks in Ruby and vice-versa.

In terms of debugging support, *SQUIMERA*'s debugger could be extended to provide similar features that other and more mature IDEs provide. This includes, for instance, more options that allow developers to drill down on certain exceptions. In addition, the detection of unhandled exceptions could be improved further, so that it covers more exception handling scenarios.

Currently, we have mainly focused on run-time tools in *SQUIMERA*. But we want to adapt more Smalltalk tools, such as the system browser or the test runner which can be used to write program code. With the system browser, it could be possible to build hybrid programs that use different programming languages. As an example, we could build an application that uses a Python web framework, Ruby's Regex engine, and Smalltalk's collection protocol. Furthermore, the system browser could provide refactoring tools for all integrated languages. Moreover, it would be interesting to see if we could build a tool similar to Jupyter Notebook with Vivide [103] that can provide better live programming features. The test runner could, for instance, be used to support the migration of programs from one to another or even multiple languages.

Further, we want to integrate more languages into *SQUIMERA*. Ruby and Python are relatively similar to Smalltalk, so we want to find out if we can integrate languages based on other programming paradigms in a similar way. There are many RPython-based implementations of various languages that can be used, for example for Prolog [6], Racket [8], Lisp [4], SQL [15], and many others.

Lastly, we want to investigate if our approach can also be applied to other language ecosystems. The GraalVM, for instance, also allows interoperability of different languages implemented in the Truffle framework. Hence, it would be interesting to see if it is possible to adapt tools from Eclipse or the NetBeans IDE [17] in such a way that they behave consistently for languages implemented in Truffle.

Despite the ideas for future work, *SQUIMERA* demonstrates that it is possible to reuse live programming tools of a Smalltalk environment for different high-level dynamic languages and to create a consistent programming experience. *SQUIMERA* is a prototype system, but we have many ideas how to make it better to further improve the programming experience for developers.

## References

- [1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. “RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages”. In: *Proceedings of the 2007 Symposium on Dynamic Languages*. DLS '07. Montreal, Quebec, Canada: ACM, 2007, pages 53–64. ISBN: 978-1-59593-868-8. DOI: 10.1145/1297081.1297091.
- [2] Appcelerator, Inc. *Aptana Studio*. 2017. URL: <http://www.aptana.com/> (visited on 2017-05-18).
- [3] J. Aycock. “A Brief History of Just-in-time”. In: *ACM Comput. Surv.* 35.2 (June 2003), pages 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077.
- [4] T. Baldridge. *Pixie*. 2017. URL: <http://pixielang.org/> (visited on 2017-05-22).
- [5] E. Barrett, C. F. Bolz, and L. Tratt. “Approaches to Interpreter Composition”. In: *Computer Languages, Systems & Structures* 44, Part C (2015), pages 199–217. ISSN: 1477-8424. DOI: 10.1016/j.cl.2015.03.001.
- [6] E. Barrett, C. F. Bolz, and L. Tratt. “Unipycation: A Case Study in Cross-language Tracing”. In: *Proceedings of the 7th Workshop on Virtual Machines and Intermediate Languages*. VMIL '13. Indianapolis, USA: ACM, 2013, pages 31–40. ISBN: 978-1-4503-2601-8. DOI: 10.1145/2542142.2542146.
- [7] D. W. Barron. “Scripting Languages”. In: *Encyclopedia of Computer Science*. Chichester, UK: John Wiley & Sons Ltd., 2003, pages 1553–1557. ISBN: 0-470-86412-5.
- [8] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. “Pycket: A Tracing JIT for a Functional Language”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. Volume 50. ICFP 2015. Vancouver, BC, Canada: ACM, 2015, pages 22–34. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784740.
- [9] D. M. Beazley et al. “SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++.” In: *Proceedings of the USENIX Fourth Annual Tcl/Tk Workshop*. Monterey, USA, 1996.
- [10] K. Beck. *Smalltalk Best Practice Patterns. Volume 1: Coding*. Prentice Hall, Englewood Cliffs, NJ, USA, 1997.
- [11] F. Biancuzzi and S. Warden. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. 1st. O'Reilly Media, Inc., 2009. ISBN: 978-0596515171.

- [12] A. P. Black, O. Nierstrasz, S. Ducasse, and D. Pollet. *Pharo by Example*. Lulu.com, 2010.
- [13] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. “Tracing the Meta-level: PyPy’s Tracing JIT Compiler”. In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ICPOOLPS ’09. Genova, Italy: ACM, 2009, pages 18–25. ISBN: 978-1-60558-541-3. DOI: 10.1145/1565824.1565827.
- [14] C. F. Bolz, A. Kuhn, A. Lienhard, N. D. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. “Back to the Future in One Week – Implementing a Smalltalk VM in PyPy”. In: *Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008 Revised Selected Papers*. Edited by R. Hirschfeld and K. Rose. Berlin, Heidelberg: Springer-Verlag, 2008, pages 123–139. ISBN: 978-3-540-89275-5. DOI: 10.1007/978-3-540-89275-5\_7.
- [15] C. F. Bolz, D. Kurilova, and L. Tratt. “Making an Embedded DBMS JIT-friendly”. In: *Proceedings of the 30th European Conference on Object-Oriented Programming*. Edited by S. Krishnamurthi and B. S. Lerner. Volume 56. ECOOP’16. Dagstuhl, Germany: Schloss Dagstuhl, Leibniz-Zentrum für Informatik, June 2016, 4:1–4:24. ISBN: 978-3-95977-014-9. DOI: 10.4230/LIPIcs.ECOOP.2016.4.
- [16] C. F. Bolz, M. Leuschel, and D. Schneider. “Towards a Jitting VM for Prolog Execution”. In: *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. PPDP ’10. Hagenberg, Austria: ACM, 2010, pages 99–108. ISBN: 978-1-4503-0132-9. DOI: 10.1145/1836089.1836102.
- [17] T. Boudreau, J. Glick, S. Greene, V. Spurlin, and J. Woehr. *NetBeans: The Definitive Guide: Developing, Debugging, and Deploying Java Code*. O’Reilly Media, 2002. ISBN: 978-1-449-33255-6.
- [18] G. Bracha. *Debug Mode Is the Only Mode*. Talk at the 2012 Meeting of the IFPI TC2 Working Group on Language Design. 2012. URL: <https://gbracha.blogspot.com/2012/11/debug-mode-is-only-mode.html> (visited on 2017-05-22).
- [19] G. Bracha, P. Ahe, V. Bykov, Y. Kashai, and E. Miranda. *The Newspeak Programming Platform*. 2008. URL: <http://bracha.org/newspeak.pdf> (visited on 2017-07-01).
- [20] Brainwy Software Ltda. *PyDev*. 2017. URL: <http://www.pydev.org/> (visited on 2017-05-18).
- [21] G. Brandl and Pygments contributors. *Pygments – Python Syntax Highlighter*. 2014. URL: <http://pygments.org/> (visited on 2017-05-16).
- [22] P. A. Buhr. “Advanced Exception Handling Mechanisms”. In: *IEEE Transactions on Software Engineering* 26.9 (Sept. 2000), pages 820–836. ISSN: 0098-5589. DOI: 10.1109/32.877844.

## References

- [23] P. Butterworth, A. Otis, and J. Stein. “The GemStone Object Database Management System”. In: *Commun. ACM* 34.10 (Oct. 1991), pages 64–77. ISSN: 0001-0782. DOI: 10.1145/125223.125254.
- [24] P. Carbonnelle. *Top IDE index*. 2016. URL: <http://pypl.github.io/IDE.html> (visited on 2017-05-18).
- [25] L. A. Chamberland, S. F. Lymer, and A. G. Ryman. “IBM VisualAge for Java”. In: *IBM Syst. J.* 37.3 (July 1998), pages 386–408. ISSN: 0018-8670. DOI: 10.1147/sj.373.0386.
- [26] T. E. Cheatham Jr. “Motivation for Extensible Languages”. In: *SIGPLAN Not.* 4.8 (Aug. 1969), pages 45–49. ISSN: 0362-1340. DOI: 10.1145/1115858.1115869.
- [27] A. Cleary, S. Kohn, S. G. Smith, and B. Smolinski. “Language Interoperability Mechanisms for High-performance Scientific Applications”. In: *Proceedings of the 1998 SIAM Workshop on Object-Oriented Methods for Interoperable Scientific and Engineering Computing*. Volume 99. SIAM, July 1999, pages 30–39. ISBN: 0-89871-445-1.
- [28] I. Craig. *The Interpretation of Object-Oriented Programming Languages*. 2nd. Springer-Verlag New York, Inc., 2001. ISBN: 978-1-852-33547-2.
- [29] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. “Compiling Java Just in Time”. In: *IEEE Micro* 17.3 (1997), pages 36–43. ISSN: 0272-1732. DOI: 10.1109/40.591653.
- [30] W. Cunningham. *Visual Age Java Gripes*. 2009. URL: <http://wiki.c2.com/?VisualAgeJavaGripes> (visited on 2017-05-18).
- [31] T. De Smedt, L. Nijs, and W. Daelemans. “Creative Web Services with Pattern”. In: *Proceedings of the 5th International Conference on Computational Creativity*. ICCO 2014. Ljubljana, Slovenia, 2014. ISBN: 978-961-264-055-2.
- [32] J. Des Rivières and J. Wiegand. “Eclipse: A Platform for Integrating Development Tools”. In: *IBM Syst. J.* 43.2 (Apr. 2004), pages 371–383. ISSN: 0018-8670. DOI: 10.1147/sj.432.0371.
- [33] L. P. Deutsch and A. M. Schiffman. “Efficient Implementation of the Smalltalk-80 System”. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’84. Salt Lake City, USA: ACM, 1984, pages 297–302. ISBN: 0-89791-125-3. DOI: 10.1145/800017.800542.
- [34] L. Diekmann and L. Tratt. “Eco: A Language Composition Editor”. In: *Proceedings of the 7th International Conference on Software Language Engineering*. Edited by B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju. SLE 2014. Springer International Publishing, 2014, pages 82–101. ISBN: 978-3-319-11245-9. DOI: 10.1007/978-3-319-11245-9\_5.
- [35] M. Dmitriev. “Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation”. In: *Proceedings of the 4th International Workshop on Software and Performance*. WOSP ’04. Redwood Shores, USA: ACM, 2004, pages 139–150. ISBN: 1-58113-673-0. DOI: 10.1145/974044.974067.

- [36] M. Eaddy and S. Feiner. *Multi-language Edit-and-continue for the Masses*. Technical report CUCS-01505. Columbia University Department of Computer Science, 2005.
- [37] Y. Endoh. *Optcarrot: A NES Emulator for Ruby Benchmark*. 2016. URL: <https://github.com/mame/optcarrot> (visited on 2017-05-05).
- [38] M. Fayad and D. C. Schmidt. "Object-oriented Application Frameworks". In: *Commun. ACM* 40.10 (Oct. 1997), pages 32–38. ISSN: 0001-0782. DOI: 10.1145/262793.262798.
- [39] T. Felgentreff. "The Design and Implementation of Object-Constraint Programming". PhD thesis. 2017.
- [40] T. Felgentreff. *Topaz Ruby*. <http://lanyrd.com/2013/wrocloverb/sccygw/>, <https://github.com/topazproject/topaz>. Invited Talk at the 2013 edition of Wroclove.rb. Mar. 2013.
- [41] T. Felgentreff, T. Pape, P. Rein, and R. Hirschfeld. "How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain". In: *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies. IWST '16*. Prague, Czech Republic: ACM, 2016, 21:1–21:10. ISBN: 978-1-4503-4524-8. DOI: 10.1145/2991041.2991062.
- [42] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. First. O'Reilly, 2008. ISBN: 978-0596516178.
- [43] Free Software Foundation, Inc. *GNU Smalltalk*. 2015. URL: <http://smalltalk.gnu.org/> (visited on 2017-07-01).
- [44] B. Freudenberg, Y. Ohshima, and S. Wallace. "Etoys for One Laptop Per Child". In: *Proceedings of the 7th International Conference on Creating, Connecting and Collaborating through Computing. C5'09*. IEEE. Jan. 2009, pages 57–64. DOI: 10.1109/C5.2009.9.
- [45] N. H. Gehani. "Exceptional C or C with exceptions". In: *Software: Practice and Experience* 22.10 (1992), pages 827–848. ISSN: 1097-024X. DOI: 10.1002/spe.4380221003.
- [46] GitHub Inc. *Atom Editor*. 2017. URL: <https://atom.io/> (visited on 2017-05-18).
- [47] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. "Debugging in the (Very) Large: Ten Years of Implementation and Experience". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. SOSP '09*. Big Sky, USA: ACM, 2009, pages 103–116. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629586.
- [48] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. The Red Book. Boston, MA, USA: Addison-Wesley Longman, 1984. ISBN: 978-0-201-11372-3.

## References

- [49] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. The Blue Book. Boston, MA, USA: Addison-Wesley Longman, 1983. ISBN: 978-0201113716.
- [50] J. Goldsmith. *Python Library Wikipedia*. 2014. URL: <https://wikipedia.readthedocs.io/en/latest/> (visited on 2017-05-17).
- [51] A. Green. *libffi*. 2017. URL: <http://sourceware.org/libffi> (visited on 2017-05-22).
- [52] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck. “High-performance Cross-language Interoperability in a Multi-language Runtime”. In: *SIGPLAN Not.* 51.2 (Oct. 2015), pages 78–90. ISSN: 0362-1340. DOI: 10.1145/2936313.2816714.
- [53] M. Grimmer, C. Seaton, T. Würthinger, and H. Mössenböck. “Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages”. In: *Proceedings of the 14th International Conference on Modularity*. Fort Collins, CO, USA: ACM, 2015, pages 1–13. ISBN: 978-1-4503-3249-1. DOI: 10.1145/2724525.2728790.
- [54] M. Grinberg. *Flask Web Development: Developing Web Applications with Python*. 1st. O’Reilly Media, Inc., 2014. ISBN: 978-1449372620.
- [55] J. Hamilton. “Language Integration in the Common Language Runtime”. In: *SIGPLAN Not.* 38.2 (Feb. 2003), pages 19–28. ISSN: 0362-1340. DOI: 10.1145/772970.772973.
- [56] M. Hlopko, J. Kurš, J. Vraný, and C. Gittinger. “On the Integration of Smalltalk and Java: Practical Experience with STX:LIBJAVA”. In: *Proceedings of the International Workshop on Smalltalk Technologies*. IWST ’12. Ghent, Belgium: ACM, 2012, 5:1–5:12. ISBN: 978-1-4503-1897-6. DOI: 10.1145/2448963.2448968.
- [57] U. Hölzle, C. Chambers, and D. Ungar. “Optimizing Dynamically-typed Object-oriented Languages with Polymorphic Inline Caches”. In: *Proceeding of the European Conference on Object-Oriented Programming*. Edited by P. America. ECOOP’91. Berlin, Heidelberg: Springer-Verlag, July 1991, pages 21–38. ISBN: 978-3-540-47537-8. DOI: 10.1007/BFb0057013.
- [58] T. Hopkins and B. Horan. *Smalltalk: An Introduction to Application Development Using VisualWorks*. Hertfordshire, UK: Prentice Hall International (UK) Ltd., 1995. ISBN: 0-13-318387-4.
- [59] 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: *SIGPLAN Not.* 32.10 (Oct. 1997), pages 318–326. ISSN: 0362-1340. DOI: 10.1145/263700.263754.
- [60] JetBrains. *PyCharm*. 2017. URL: <https://www.jetbrains.com/pycharm/> (visited on 2017-05-18).
- [61] JetBrains. *RubyMine*. 2017. URL: <https://www.jetbrains.com/ruby/> (visited on 2017-05-18).

- [62] A. Kay. *Squeak Etoys, Children & Learning*. 2005. URL: [http://vpri.org/pdf/rn2005001\\_learning.pdf](http://vpri.org/pdf/rn2005001_learning.pdf) (visited on 2017-07-01).
- [63] D. Kirkby. *live-py-plugin*. 2017. URL: <http://donkirkby.github.io/live-py-plugin/> (visited on 2017-05-20).
- [64] D. E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89683-4.
- [65] G. Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0-201-11669-3.
- [66] S. Lantinga. *Simple DirectMedia Layer*. 2016. URL: <https://www.libsdl.org> (visited on 2017-05-05).
- [67] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999. ISBN: 978-0201325775.
- [68] M. D. MacLaren. "Exception Handling in PL/I". In: *SIGOPS Oper. Syst. Rev.* 11.2 (Mar. 1977), pages 101–104. ISSN: 0163-5980. DOI: 10.1145/390018.808316.
- [69] J. Mair. *Pry – An IRB Alternative and Runtime Developer Console*. 2016. URL: <http://pryrepl.org/> (visited on 2017-05-20).
- [70] J. Maloney. *Morphic: The Self User Interface Framework*. 1995. URL: <http://ftp.squeak.org/docs/Self-4.0-UI-Framework.pdf> (visited on 2017-07-01).
- [71] J. McCarthy. "History of Programming Languages I". In: edited by R. L. Wexelblat. New York, NY, USA: ACM, 1981. Chapter History of LISP, pages 173–185. ISBN: 0-12-745040-8. DOI: 10.1145/800025.1198360.
- [72] MediaWiki.org. *API: Main Page*. 2002. URL: [https://www.mediawiki.org/wiki/API:Main\\_page](https://www.mediawiki.org/wiki/API:Main_page) (visited on 2017-05-17).
- [73] B. Meyer. *Object-Oriented Software Construction*. 2nd. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0-13-629049-3.
- [74] Microsoft. *Edit and Continue*. 2016. URL: <https://docs.microsoft.com/en-us/visualstudio/debugger/edit-and-continue> (visited on 2017-05-24).
- [75] Microsoft. *Visual Studio*. 2017. URL: <https://www.visualstudio.com/> (visited on 2017-05-18).
- [76] E. Miranda. "The Cog Smalltalk Virtual Machine". In: *Proceedings of the 5th Workshop on Virtual Machines and Intermediate Languages*. VMIL '11. ACM, 2011.
- [77] E. Miranda and contributors. *OpenSmalltalkVM – Cross-platform Virtual Machine for Squeak, Pharo, Cuis, and Newspeak*. 2017. URL: <https://github.com/OpenSmalltalk/opensmalltalk-vm> (visited on 2017-05-22).
- [78] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

## References

- [79] O. Nierstrasz, S. Ducasse, and D. Pollet. *Squeak by Example*. Square Bracket Associates, 2009. ISBN: 978-3-9523341-0-2.
- [80] C. O. Nutter, T. Enebo, N. Sieger, O. Bini, and I. Dees. *Using JRuby: Bringing Ruby to Java*. 1st. Pragmatic Bookshelf, 2011. ISBN: 978-1934356654.
- [81] D. Pacheco. “Postmortem Debugging in Dynamic Environments”. In: *Queue* 9.10 (Oct. 2011), 12:10–12:21. ISSN: 1542-7730. DOI: 10.1145/2039359.2039361.
- [82] M. Pall. *The LuaJIT Project*. 2005. URL: <http://luajit.org/> (visited on 2017-04-20).
- [83] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. “Microservices in Practice, Part 1: Reality Check and Service Design”. In: *IEEE Softw.* 34.1 (Jan. 2017), pages 91–98. ISSN: 0740-7459. DOI: 10.1109/MS.2017.24.
- [84] Python Software Foundation. *ctypes – A Foreign Function Library for Python*. 2017. URL: <https://docs.python.org/2/library/ctypes.html> (visited on 2017-05-22).
- [85] Python Software Foundation. *pdb – The Python Debugger*. 2017. URL: <https://docs.python.org/2/library/pdb.html> (visited on 2017-05-20).
- [86] P. Rein, S. Lehmann, T. Mattis, and R. Hirschfeld. “How Live Are Live Programming Systems? Benchmarking the Response Times of Live Programming Environments”. In: *Proceedings of the Programming Experience 2016 (PX/16) Workshop*. PX/16. Rome, Italy: ACM, 2016, pages 1–8. ISBN: 978-1-4503-4776-1. DOI: 10.1145/2984380.2984381.
- [87] L. Renggli, T. Gîrba, and O. Nierstrasz. “Embedding Languages without Breaking Tools”. In: *Proceedings of the 24th European Conference on Object-Oriented Programming*. Edited by T. D’Hondt. ECOOP’10. Berlin, Heidelberg: Springer-Verlag, June 2010, pages 380–404. ISBN: 978-3-642-14107-2. DOI: 10.1007/978-3-642-14107-2\_19.
- [88] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. “Scratch: Programming for All”. In: *Commun. ACM* 52.11 (Nov. 2009), pages 60–67. ISSN: 0001-0782. DOI: 10.1145/1592761.1592779.
- [89] J. C. Reynolds. “The Discoveries of Continuations”. In: *Lisp Symb. Comput.* 6.3-4 (Nov. 1993), pages 233–248. ISSN: 0892-4635. DOI: 10.1007/BF01019459.
- [90] M. Richards. *Bench*. 1999. URL: <https://www.cl.cam.ac.uk/~mr10/Bench.html> (visited on 2017-05-18).
- [91] M. Rigger, M. Grimmer, C. Wimmer, T. Würthinger, and H. Mössenböck. “Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle”. In: *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*. VMIL ’16. Amsterdam, Netherlands: ACM, 2016, pages 6–15. ISBN: 978-1-4503-4645-0. DOI: 10.1145/2998415.2998416.

- [92] A. Rigo and S. Pedroni. “PyPy’s approach to virtual machine construction”. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. OOPSLA ’06. Portland, USA: ACM, 2006, pages 944–953. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176753.
- [93] Ruby Community. *Class: DEBUGGER\_\_*. 2017. URL: [http://ruby-doc.org/stdlib-2.4.1/libdoc/debug/rdoc/DEBUGGER\\_\\_.html](http://ruby-doc.org/stdlib-2.4.1/libdoc/debug/rdoc/DEBUGGER__.html) (visited on 2017-05-20).
- [94] Ruby Community. *Module: Fiddle*. 2017. URL: <http://ruby-doc.org/stdlib-2.4.1/libdoc/fiddle/rdoc/Fiddle.html> (visited on 2017-05-22).
- [95] C. Seaton, M. L. Van De Vanter, and M. Haupt. “Debugging at Full Speed”. In: *Proceedings of the Workshop on Dynamic Languages and Applications*. Dyla’14. Edinburgh, UK: ACM, 2014, 2:1–2:13. ISBN: 978-1-4503-2916-3. DOI: 10.1145/2617548.2617550.
- [96] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. “Virtual Machine Showdown: Stack Versus Registers”. In: *ACM Trans. Archit. Code Optim.* 4.4 (Jan. 2008), 2:1–2:36. ISSN: 1544-3566. DOI: 10.1145/1328195.1328197.
- [97] J. Skinner. *Sublime Text*. 2017. URL: <https://www.sublimetext.com/> (visited on 2017-05-18).
- [98] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 978-1558609105.
- [99] M. Springer. “Inter-language Collaboration in an Object-oriented Virtual Machine”. In: *CoRR abs/1606.03644* (2016).
- [100] R. M. Stallman. “EMACS the Extensible, Customizable Self-documenting Display Editor”. In: *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*. Portland, USA: ACM, 1981, pages 147–156. ISBN: 0-89791-050-8. DOI: 10.1145/800209.806466.
- [101] B. Stewart. *An Interview with the Creator of Ruby*. 2011. URL: <http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html> (visited on 2017-04-25).
- [102] M. Taeumel and R. Hirschfeld. “Evolving User Interfaces From Within Self-supporting Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs”. In: *Proceedings of the Programming Experience 2016 (PX/16) Workshop*. PX/16. Rome, Italy: ACM, 2016, pages 43–59. ISBN: 978-1-4503-4776-1. DOI: 10.1145/2984380.2984386.
- [103] M. Taeumel, B. Steinert, and R. Hirschfeld. “The VIVIDE Programming Environment: Connecting Run-time Information with Programmers’ System Knowledge”. In: *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2012.

## References

- Tucson, USA: ACM, 2012, pages 117–126. ISBN: 978-1-4503-1562-3. DOI: 10.1145/2384592.2384604.
- [104] S. L. Tanimoto. “A Perspective on the Evolution of Live Programming”. In: *2013 1st International Workshop on Live Programming (LIVE)*. May 2013, pages 31–34. DOI: 10.1109/LIVE.2013.6617346.
- [105] The Eclipse Foundation. *Dynamic Languages Toolkit*. 2017. URL: <http://www.eclipse.org/dltk/> (visited on 2017-05-18).
- [106] The PyPy Project. *Application-level Stackless Features*. 2017. URL: <http://doc.pypy.org/en/latest/stackless.html> (visited on 2017-05-10).
- [107] TIOBE Software. *TIOBE Programming Community Index for April 2017*. 2017. URL: <https://www.tiobe.com/tiobe-index/> (visited on 2017-04-25).
- [108] M. L. Van De Vanter. “Building Debuggers and Other Tools: We Can “Have It All””. In: *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. IC00OLPS ’15. Prague, Czech Republic: ACM, 2015, 2:1–2:3. ISBN: 978-1-4503-3657-4. DOI: 10.1145/2843915.2843917.
- [109] G. van Rossum. *Python Library Reference*. Technical report CS-R9524. Centrum voor Wiskunde en Informatica (CWI), 1995.
- [110] G. van Rossum. *Python Reference Manual*. Technical report CS-R9525. Centrum voor Wiskunde en Informatica (CWI), 1995.
- [111] G. van Rossum. *Python Tutorial*. Technical report CS-R9526. Centrum voor Wiskunde en Informatica (CWI), 1995.
- [112] B. Victor. *Stop Drawing Dead Fish*. Talk to the San Francisco ACM SIGGRAPH. May 12, 2012. URL: <http://san-francisco.siggraph.org/stop-drawing-dead-fish/> (visited on 2017-07-01).
- [113] J. Vraný, J. Karpíšek, and P. Špaček. *SmallRuby Project Website*. 2010. URL: <https://swing.fit.cvut.cz/projects/smallruby> (visited on 2017-05-18).
- [114] J. Vraný and M. Píše. “Multilanguage Debugger Architecture”. In: *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*. SOFSEM ’10. Berlin, Heidelberg: Springer-Verlag, 2010, pages 731–742. ISBN: 978-3-642-11266-9. DOI: 10.1007/978-3-642-11266-9\_61.
- [115] C. Wimmer and T. Würthinger. “Truffle: A Self-optimizing Runtime System”. In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH ’12. Tucson, USA: ACM, 2012, pages 13–14. ISBN: 978-1-4503-1563-0. DOI: 10.1145/2384716.2384723.
- [116] Wingware. *Wing Python IDE*. 2017. URL: <http://wingware.com/> (visited on 2017-05-18).

- [117] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. “One VM to Rule Them All”. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, USA: ACM, 2013, pages 187–204. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581.
- [118] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. “Self-optimizing AST Interpreters”. In: *Proceedings of the 8th Symposium on Dynamic Languages*. DLS '12. Tucson, USA: ACM, 2012, pages 73–82. ISBN: 978-1-4503-1564-7. DOI: 10.1145/2384577.2384587.
- [119] S. Yemini and D. M. Berry. “A Modular Verifiable Exception Handling Mechanism”. In: *ACM Trans. Program. Lang. Syst.* 7.2 (Apr. 1985), pages 214–243. ISSN: 0164-0925. DOI: 10.1145/3318.3320.
- [120] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1-558-60866-4.
- [121] D. Zingaro. *Modern extensible languages*. Technical report 47. McMaster University, 2007.



# Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
119	978-3-86956-406-7	<b>k-Inductive invariant Checking for Graph Transformation Systems</b>	Johannes Dyck, Holger Giese
118	978-3-86956-405-0	<b>Probabilistic timed graph transformation systems</b>	Maria Maximova, Holger Giese, Christian Krause
117	978-3-86956-401-2	<b>Proceedings of the Fourth HPI Cloud Symposium "Operating the Cloud" 2016</b>	Stefan Klauck, Fabian Maschler, Karsten Tausche
116	978-3-86956-397-8	<b>Die Cloud für Schulen in Deutschland : Konzept und Pilotierung der Schul-Cloud</b>	Jan Renz, Catrina Grella, Nils Karn, Christiane Hagedorn, Christoph Meinel
115	978-3-86956-396-1	<b>Symbolic model generation for graph properties</b>	Sven Schneider, Leen Lambers, Fernando Orejas
114	978-3-86956-395-4	<b>Management Digitaler Identitäten : aktueller Status und zukünftige Trends</b>	Christian Tietz, Chris Pelchen, Christoph Meinel, Maxim Schnjakin
113	978-3-86956-394-7	<b>Blockchain : Technologie, Funktionen, Einsatzbereiche</b>	Tatiana Gayvoronskaya, Christoph Meinel, Maxim Schnjakin
112	978-3-86956-391-6	<b>Automatic verification of behavior preservation at the transformation level for relational model transformation</b>	Johannes Dyck, Holger Giese, Leen Lambers
111	978-3-86956-390-9	<b>Proceedings of the 10th Ph.D. retreat of the HPI research school on service-oriented systems engineering</b>	Christoph Meinel, Hasso Plattner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich, Emmanuel Müller
110	978-3-86956-387-9	<b>Transmorphic : mapping direct manipulation to source code transformations</b>	Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, Robert Hirschfeld
109	978-3-86956-386-2	<b>Software-Fehlerinjektion</b>	Lena Feinbube, Daniel Richter, Sebastian Gerstenberg, Patrick Siegler, Angelo Haller, Andreas Polze
108	978-3-86956-377-0	<b>Improving Hosted Continuous Integration Services</b>	Christopher Weyand, Jonas Chromik, Lennard Wolf, Steffen Kötte, Konstantin Haase, Tim Felgentreff, Jens Lincke, Robert Hirschfeld





ISBN 978-3-86956-422-7  
ISSN 1613-5652