Calico: An Ecosystem for Dynamic Languages

Douglas S. Blank

Bryn Mawr College
Department of Computer Science
Bryn Mawr, PA 19010 USA
dblank@cs.brynmawr.edu

Mark F. Russo, Ph.D.

Rowan University
Department of Computer Science
Glassboro, NJ 08028 USA
russomf@gmail.com

James B. Marshall

Sarah Lawrence College
Department of Computer Science
Bronxville, NY 10708 USA
jmarshall@slc.edu

Abstract

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: interpreters, runtime environments

General Terms Languages, Experimentation

Keywords Integrated Development Environment, dynamic languages, virtual machine, language implementation, language interoperations

1. Overview

The Calico Project defines a collection of technologies designed to create an ecosystem for dynamic languages. The Calico ecosystem has a number of interesting properties: 1) allows multiple dynamic languages to interact and interoperate; 2) libraries written for the ecosystem are usable by all of the languages as if they were native to each language; 3) provides a smooth continuum of experience for the beginning computer programmer; 4) exposes a unified interface for exploring a variety of dynamic languages; and 5) is an open-source testbed for developing new concepts in dynamic languages. As such, Calico defines a useful ecosystem for students, educators, and developers wishing to explore computer science and pedagogy.

Calico is defined by three layers of technologies: a universal virtual computer, a dynamic language runtime, and support libraries and executables. In addition, an Integrated Development Environment (IDE) pulls all of these pieces together, for students and researchers. As implemented, Calico languages can all live and operate in the same space, share variables and code, and the code can run competitively fast. The following sections go into the details of each of the pieces that comprise the Calico ecosystem.

1.1 Universal Virtual Computer

At the foundation of Calico is what could be characterised from the programmer's perspective as a universal virtual computer. The dynamic language programmer should be able to write code capable of modern functionality without having to worry about the details of the operating system or underlying hardware. For example, the programmer should be able to write programs capable of: turning text to speech; playing audio files; defining functions that can be used for tone generation; creating graphical user interfaces

with unconstrained drawing abilities; reading and writing standard image formats; and providing access to modern input/output hardware devices, such as gamepads, joysticks, webcams, and robots. In addition, such a system should be easy to install and maintain, with limited need for platform-specific dependencies. Finally, we believe that a long term platform for research and education should be free, both in terms of price and ability to distribute and alter. This section examine the components of a universal virtual computer.

1.1.1 Virtual Machine

A large portion of the universal virtual computer is defined by a "process virtual machine." There are a number of process virtual machines (VMs) one could use for a dynamic language ecosystem. We wanted a VM with a large and active development community, and so excluded more narrowly-focused VMs such as the Squeak Virtual Machine [18]. Oracle's Java and Microsoft's .NET are two VMs that both define viable possibilities: both have object-oriented programming languages, compilers, and associated virtual machines and runtimes. The Java VM (JVM) has the Java language (among other possibilities), and .NET has C# (among other possibilities). Both systems compile to bytecode that is executed by their respective virtual machines' runtime.

Although both VMs have served as the foundation for dynamic languages, Java does not have a complete and robust official open-source implementation. On the other hand, .NETs virtual machine components, called the Common Language Infrastructure (CLI), have been clearly defined in a pair of Ecma standards, specifically Ecma #334 and Ecma #335 [21], and is protected by a promise from Microsoft not to sue [23]. More importantly, these standards have been implemented independently by Mono as open source. We have put a high value on having a complete and open source "stack" of robust software layers. Although one could debate which system is more "open", we decided to select the CLI as implemented by Mono [22]. we will refer to Mono's implementation of the CLI as MVM (for Mono Virtual Machine).

However, we are not limited to only MVM-based technologies. Using IKVM (described below) we are able to utilize code compiled for the Java VM. In this manner, we have assembled components that bring together the best of both of these two virtual machine systems. In effect, we have a "universal virtual machine" that can use a wide variety of available open-source libraries on a varied set of operating systems. One such library is the Dynamic Language Runtime described in the next section.

1.1.2 Dynamic Language Runtime

Although the MVM and the JVM both define everything necessary to implement a dynamic language, neither has built-in support for allowing such dynamic languages to interoperate. That is, even though a variety of languages can be compiled to either VM, there is little common infrastructure in the VM itself. Without a dynamic

language infrastructure, it is difficult for languages defined in the VM to define dynamic objects, and to share environments, data structures, objects, or functions. Two Java Specification Requests have been made to add such functionality to Java: JSR-223 would add language hosting and JSR-292 would add dynamic binding [13]. In fact, there are a number of projects underway to make the JVM more flexible for use with dynamic languages, including: the Da Vinci Machine [1] as the reference implementation of JSR-292; Project Lambda [2] adds lambda to the JVM; Project Jigsaw adds supported for importing versioned modules [3]; and Nashorn [4] adds caches for dynamic invokes. When complete, these technologies will extend the Java technologies to make them much more suitable for dynamic languages.

On the other hand, the MVM has an existing, mature framework for implementing languages called the Dynamic Language Runtime (DLR) [5], and at least two dynamic languages have been written using the DLR: IronPython [6], IronRuby [7], and PowerShell [30]. The DLR contains many tools and technologies for language writers to create languages, and abilities for the languages to interoperate. Calico includes the DLR, and contains versions of IronPython and IronRuby.

The DLR provides the following [26]:

- A dynamic, shared type system (aka, DynamicObject)
- Dynamic method dispatch
- Dynamic code generation
- Hosting API

The DLR contains all the support necessary for creating, parsing, and executing modern dynamic languages. However, even with the DLR and the virtual machine, there are still a large set of functions that a modern programmer would expect from a universal computer (including audio and graphics). We now explore these supporting libraries.

1.2 Supporting Libraries

To create a universal virtual computer, we need to have a set of capabilities on top of the virtual machine and dynamic language runtime that run the gamut from audio to graphics. Of course, operating systems and hardware vary wildly. Fortunately, there are open source libraries for the MVM that cover these requirements. For graphics, we selected Gtk# [8] which wraps low-level C-based libraries. For accessing audio and hardware devices (such as gamepads), we selected the Simple DirectMedia Layer (SDL) [9]. SDL is quite popular in open source, providing additional capabilities for many projects, including pygame [11]. Finally, we also added espeak [24] to provide cross-platform text-to-speech functionality.

The MVM and the above mentioned libraries form the lowest level of the virtual computer. These libraries are machine-dependent, and, as such, must be compiled for each platform. We attempted to create the fewest number of machine-specific dependencies to make the virtual computer easy to maintain and port to new architectures. Finally, after this low-level set of dependencies are satisfied, then additional libraries can be written in a machine-independent fashion using the MVM, or with the JVM and converted with IKVM.

IKVM is an interesting open source project that translates binary code between MVM/.NET and the JVM [12]. It has the ability to make code compiled for the JVM availble to the MVM, and, likewise, to make code compiled for the MVM (or .NET in general) available to the JVM. We see an example of IKVM with Calico Java (discussed below).

Combined with the MVM and DLR, these libraries complete the functionality of a universal virtual computer so that one can write

programs that utilize graphics user interfaces (including widgets, freestyle pixel manipulation, event handlers, and image formats), audio (including text-to-speech and tone generation), access to hardware devices, and other modern functionality. One can then write in dynamic languages with the full scope of abilities on most of the operating systems in use, including Windows, Linux, Mac OSX, and FreeBSD.

1.3 Integrated Development Environment

From the user's perspective, an important part of using any programming language is the user interface. The Calico IDE is designed to be agnostic to any particular language, or particular use.

2. Languages

[FIXME] What defines a Calico language? Document, Engine

First-class languages: stepper, breakpoints, share variables, call functions, execute/evaluate code and expressions from other languages, import Calico modules as if they were native.

Second-class: naive ports, minimal interoperation, possibly unable to import modules.

Used for education [15, 16].

2.1 Calico Jigsaw

Block-based programming languages have facilitated our ability to introduce computer programming to novice programmers. Perhaps the most notable example of this is Scratch [http://scratch.mit.edu]. Building executable programs by plugging together virtual blocks is a concept that is much less intimidating to a beginner, especially when compared with the precise syntactical rules demanded by many modern programming languages. For this reason we sought to include a block-based language as part of the Calico ecosystem. We named our block language Calico Jigsaw.

The Jigsaw editor provides a palette of blocks that are dragged onto a canvas using the mouse and connected to form an executable program (Figure 1). Jigsaw blocks may have properties that can be assigned values by the programmer. For example, the Jigsaw "repeat" block has a Repetitions property that can be assigned a numeric value indicating the number of times an inner stack of blocks should be executed. But beyond the standard functionality expected in any block language, Jigsaw has several additional features that we feel make it unique.
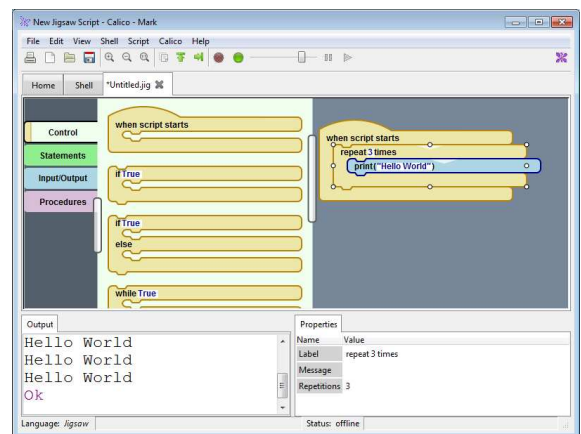

Figure 1. Jigsaw in action.

The first feature worth noting is that Jigsaw is built as a wrapper around IronPython. Even though Jigsaw is written in C#, most Jigsaw statement blocks are implemented as Python statements. This integration between C# and IronPython is possible with help from

the DLR. Block statements are compiled and held as compiled code objects that are ready to be evaluated when the Jigsaw program runs. A consequence of this design decision is that Python syntax often can be used directly in Jigsaw. For example, instead of entering primitive values for block properties, Python expressions can be substituted directly. The IfTest conditional statement in a Jigsaw "if" block can be a boolean expression written using Python syntax (Figure 2). We believe that this approach better prepares the student for a smooth transition to higher level languages.
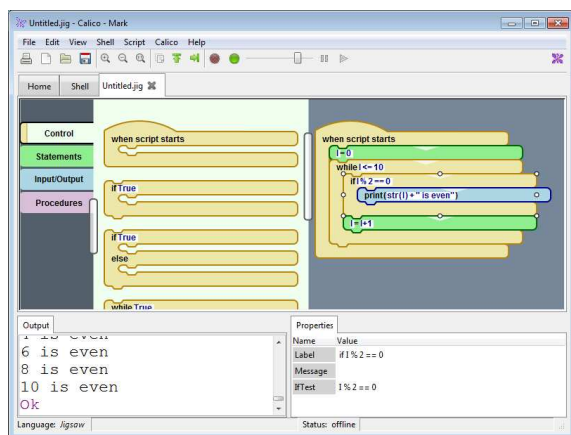
Figure 2. Jigsaw blocks leverage IronPython expressions.

Although Jigsaw statements and expressions are borrowed from IronPython, Jigsaw implements its own program control structures. Blocks that provide looping, conditionals and procedures are all implemented entirely in Mono C#. Program execution is managed using Jigsaw's own implementation of call stacks and stack frames along with local and global namespaces, an idea that is borrowed from Python.

Another unique feature of Jigsaw is that it has a form of multitasking baked in to its execution engine. A Jigsaw program begins execution at a "when script starts" block, and proceeds down the attached stack of blocks, executing each in sequence. But a Jigsaw program can contain any number of "when script starts" blocks, which all begin execution at the same time. As a result, multiple block stacks a Jigsaw program can execute in parallel. See Figure 3 for an example. Blocks colors white are currently being executed.
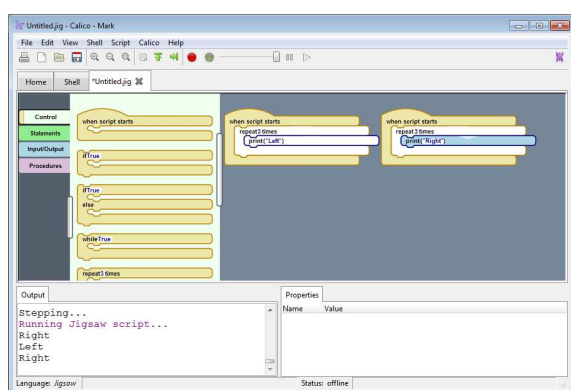

Figure 3. A Jigsaw program running multiple block stacks in parallel.

The parallelism implemented in Jigsaw is a form of cooperative multitasking, not the more common preemptive style. Jigsaw executes multiple block stacks in a round-robin fashion. It is important

to note that the execution of one block does not necessarily complete before the Jigsaw engine switches to a block in another stack. Indeed, multiple blocks in different stacks can execute at the same time. This is possible because block execution is implemented as a C# Enumerator, which is carefully designed for each block type to yield at safe places throughout its execution. With this approach a novice programmer can assemble a program that multitasks while avoiding the subtle bugs that often crop up with a true preemptive model. For example, a "background" task that continuously monitors a sensor and updates a global variable can safely coexist with a second loop that uses the value of the global variable to draw a graphic on the screen.

Calico Jigsaw allows the export of any Jigsaw script to Calico Python (described below). However, as it must convert Jigsaw's controls to those of the Python language, the semantics must also be converted. Figure 4 shows the above Jigsaw program as it appears after it has been exported to Python. Python does not have the built-in ability to run functions simultaneously as does Jigsaw. But, the Calico Myro module (described below) includes a friendly interface for using threads through the "doTogether" function, and so the export translates the Jigsaw program to the proper form. Notice, however, that the Output tab indicates that Python's threads do not guarantee even multitasking as the 'Left' and 'Right' outputs are not perfectly interleaved.
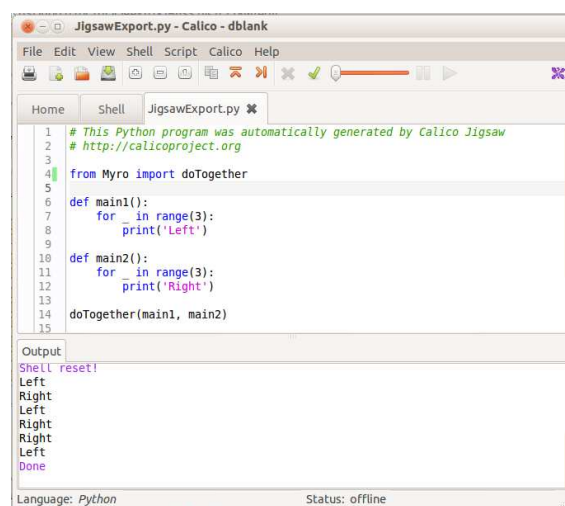
Figure 4. The Jigsaw program from Figure 3 exported as Python. Note that the blocks have been converted to functions (main1, and main2) but the output is not guaranteed to run in lockstep as it does with Jigsaw's multitasking.

2.2 Calico Python

Based on IronPython.

2.3 Calico Ruby

2.4 Calico Scheme

We wrote a properly tail-recursive implementation of Scheme from scratch and integrated it into Calico. Tail-recursion is an idiom that students often discover on their own as a way of implementing iteration [27]. Unfortunately, most of the languages commonly used in introductory courses (for example, Java, Python, and C++) do not handle tail-recursion correctly, which may lead to crashes if a student's program exceeds the language's recursion depth limit. This is seen by students as a bug in their code, whereas it is really a limitation of the underlying language implementation. Unlike other

Scheme implementations for the MVM, Calico Scheme does not rely on the underlying call stack of the implementation language to manage function calls, nor does it require functions to be compiled to Common Language Runtime (CLR) bytecode. Consequently, tail calls are handled properly in Calico Scheme, as required by the Scheme language specification [31], imposing no limit on the depth of the call stack (aside from system memory limitations).

Our Scheme implementation is itself written in Scheme, utilizing full continuation-passing style (CPS). Because the interpreter is in CPS, adding support for new types of control structures or other language constructs to Calico Scheme is easy. We wrote a transformation program to automatically transform the high-level CPS code into a low-level trampolined register machine [33]. Although the resulting register machine code is still in Scheme, it does not rely on Scheme's call stack, higher-order procedures, or parameter-passing mechanisms. Thus it can be translated directly into low-level C# code and then compiled for the MVM.

Calico Scheme supports all of the main language features of standard Scheme, such as first-class procedures and continuations, but also adds several new features of it own. For example, we included support for exception handling in the form of `try`, `catch`, `finally`, and `raise` expressions. Of course, the equivalent functionality can also be easily achieved with `call/cc`, but since many students may already be familiar with exception handling constructs in Java or Python, we decided to include similar types of constructs in Calico Scheme as well. To aid in debugging when an exception is thrown, we implemented a user-configurable stack-trace reporting mechanism that mimics the appearance of stack traces generated in Python. We also included support for McCarthy's nondeterministic choice operator `amb` [32], called `choose` in Calico Scheme. This allows students to easily explore the very different—and very powerful—paradigm of nondeterministic computing, to which most of them have probably not been exposed before.

2.5 Experimental Languages

To add a language to Calico, one need only define a single function that returns a class instance. In fact, Calico comes with a fill-in-the-blank wizard for creating new text-based languages. For text-based languages, one need only provide the language's name, a string for beginning line comments, and the the language's source code file extension (such as ".py" for the Python language). Further refinements can help make the language more useful for programmers. Such refinements include providing language keyword details for the IDE's color syntax highlighting, and, of course, the language's parser and interpreter. Because Calico supports language interoperations, new Calico languages can be written in existing Calico languages. For example, Calico's Basic and Logo languages are themselves written in Calico Python. As another example, Calico Java's language definition is written in Python, but the Java interpreter is written in Java (described in more detail below).

One might want to use the Calico infrastructure for testing out a new language. For example, if you had an interpreter (written in any Calico language, or converted to use the MVM or JVM) then you could put a light wrapper around it, and simple use the Calico IDE to make it easy to use. If you list the keywords, and a few other items, you could also have color syntax highlighting. If you add hooks to your language to import Calico modules, then your experimental language would have access to all of the included libraries (described below). Finally, if you add a method to provide line numbers, and intercept breakpoints, then your language would also be a first-class Calico language.

Because it is so easy to add a new language to Calico (given that such a language might initially be a second-class language with little connection to the rest of Calico) there are a few languages that have been initiated.

2.5.1 Calico Java

Calico Java is an example of a second-class language in the ecosystem.

Java interpreter from DrJava written in Java. Java bytecode converted to MVM bytecode. Thus, can parse and interpret Java in the MVM. Even more interesting, the Calico modules can be used in the Java.

2.5.2 Calico Basic, Logo

Written in Calico Python.

2.6 Performance

Compared to other dynamic languages, the performances of Calico languages ranges from competitive to slow [20]. Likewise, between Calico languages, there is also large performance variability. Some of these differences can be mitigated over time with refinements at various levels of implementation. For example, Calico Scheme has yet to be converted to use the DLR AST. Once that transformation is in place, we expect a 10 to 100-fold increase in performance. Other differences are likely to remain, however. For example, Calico Basic is written in Calico Python, so speed improvements will probably be minimal.

Language	Time (in Seconds), Factorial of 1000
Calico Scheme (non-recursive)	0.002
CPython (recursive)	0.003
Calico Python (recursive)	0.020
Calico Ruby (recursive)	0.024
Calico Basic (using goto)	0.053
Calico Jigsaw (recursive)	0.072
Calico Scheme (recursive)	0.368

Table 1. Performance comparison between languages on a simple function: computing the factorial of 1000.

2.7 Interoperations

First class languages can call functions and use values directly. Executing statements and expressions from another language. Sharing values between languages. Sharing code between languages.

Examples of interop.

```
1  scheme> (define pytuple
2          (calico.Evaluate
3            "lambda *args: args"
4            "python"))
5  Ok
6  scheme> (pytuple 1 2 3)
7  (1, 2, 3)
8  Ok
```

Listing 1. Defining a function `pytuple` in Calico Scheme that directly uses Calico Python's lambda to construct a Python tuple. Notice that in line 7 that the tuple's representation reflects the style from Python.

```
1  scheme> (define pylist
2          (calico.Evaluate
3            "lambda *args: list(args)"
4            "python"))
5  Ok
6  scheme> (pylist 1 2 3)
7  [1, 2, 3]
8  Ok
```

Listing 2. Defining a funtion `pylist` in Calico Scheme that creates a Python list.

For example, in Python you can call a function written in Scheme.

1. First, write a regular Scheme function

2. Call the Scheme function in a CLR-compatible function wrapper

3. Make the wrapper available in the shared environment

If one of the languages is not a first class Calico language, they can still interoperate. For example, Java is not currently a first class Calico language, but you can still execute statements, and evaluate expressions from any language that has access to the calico object. Here, Calico Python calls Java to create a variable with a particular value:

```
1  python> calico.Execute("int x = 1;", "java")
2  Ok
3  python> pyx = calico.Evaluate("x", "java")
4  Ok
5  python> print(pyx)
6  <java.lang.Integer object at 0x002B [1]>
7  Ok
8  python> print(pyx.intValue())
9  1
10 Ok
```

Listing 3. Calling Java from Python.

The value of `pyx` is actually value directly from the Java world: a java.lang.Integer object. To convert it to a MVM value depends on the specifics of the foreign system. In this case, `pyx.intValue()`.

2.8 Current Limitations and Future Work

Calico cannot currently use C-based libraries. Limited exports from one language to another. Move languages from second-class to first-class languages.

In addition, Kolling mentioned in his keynote address at SIGCSE this year that he believes that we may be ready for a "structural editors" rebirth. And this would be for experts. Paraphrasing, his idea is to blend text editing with the idea of blocks (or "structures"). We can tip our hat to the idea that "blocks" can be productive to experts too.

3. Calico Modules

A Calico "module" is a library written in such a manner that it is available to all of the Calico first-class languages. However, not only is the functionality in the module available to these languages, but the modules appear as if the are a native library to each language. For example, in Calico Python one would write "import Processing" to make the Calico Processing module (discussed below) available to Python. One could then write "Processing.window(400, 300)" in Python to create a 400x300 pixel window. In Scheme, one would write "(using "Processing")" to make the Calico Processing module available to Scheme. One could then write "(Processing.window 400 300)" in Scheme to do the same thing. Finally, in Calico Jigsaw, selecting "Use a Module; Processing" from the menu would allow the window-block to be dragged onto the Jigsaw workspace. Thus, the Processing module is written and compiled once, becomes available to all of the first-class Calico languages, and used as if it were written as a native language. Likewise, if a new first-class language is introduced into the Calico ecosystem, all of these modules can be used by the new language.

To create a Calico module, a single file is written and compiled once to a Dynamic Link Library (DLL). There are DLL files that are Windows-specific; however, these DLLs are platform-neutral and can be created on any operating system for use on any other operating system. To remain platform-neutral these DLL's must be written such that they do not take advantage of Windows-specific functionality, do not rely on lower-level platform-specific C libraries (e.g., are completely "managed"), and use a subset of all possible functionality. Currently, to be accessible to all Calico languages, we currently restrict the module to only use static class methods in a toplevel-defined class. However, this limit could be relaxed in a future Calcio to allow more flexibility (constructors, nested classes, etc). In general, any managed DLL could be fully utilized by a properly flexible, dynamic, first-class languages, but might require specific knowledge about the layout of the internal classes, methods, and namespaces. Thus, we have specified a subset of all that is possible for our own modules so that no additional knowledge or discovery is needed.

```
1  // C# ModuleName.cs compiles to ModuleName.dll
2  public class ModuleName {
3      public static int Plus(int a, int b) {
4          return (a + b);
5      }
6  }
```

Listing 4. Example module template.

If the code in Figure was compiled and placed into the Calico/modules folder, then it could be used in Python (as in the form "import ModuleName"), Scheme (as in the form "(use "ModuleName")" and in Jigsaw and all of the other Calico languages, in their respective forms. We now explore three modules that provided a variety of functionality for Calico first-class languages.

3.1 Processing

Calico Processing is a module for developing digital works of art, data visualizations, interactive applications and animations. It offers Calico programmers the option to work with the familiar Processing command set (see http://processing.org/reference). The Calico Processing module attempts to be faithful to the Processing command set, including function names, arguments, and usage. The majority of the command set has been implemented, although some differences exist.

The Processing language is a subset of Java with a wide variety of commands included for creating visualization and animations. The Calico Processing module brings the Processing command set to Calico programming languages. This module does not make use of the Java language syntax. Instead, native data types and control structures of your chosen Calico language will determine how your application is constructed.

We rely upon each of the native Calico languages for the following capabilities:

- Data structures

- Program control (conditionals, iterations, etc.)

- Files and file access

- String functions

- Objects and inheritance

Mouse and keyboard events are not handled by implementing predefined functions. The Calico Processing module raises events, which are handled by the native language event handling syntax. Events raised by the module include onMousePressed, onMouseReleased, onMouseClicked, onKeyPressed, and onKeyReleased.

Loops are implemented by handling a timer tick event named onLoop.

Certain predefined fields in Processing are implemented as functions.

- "mouseX" and "mouseY" are implemented in Calico Processing as the functions "mouseX()" and "mouseY()".

- "pmouseX" and "pmouseY" are implemented as "pmouseX()" and "pmouseY()".

- "width" and "height" are implemented as "width()" and "height()".

- "focused" and "frameCount" are implemented as "focused()" and "frameCount()".

- "key" and "keyCode" are implemented as "key()" and "keyCode()".

The result is a pixel-by-pixel faithful representation of the original Processing primitives' output. However, combined with the Calico dynamic languages, the Processing module provides functionality not available within the original Processing environment. For example, Calico allows the drag-and-drop creation of Processing art via Jigsaw, line-by-line stepping through programs, and breakpoints by using any of the Calico first-class languages.

3.2 Myro

Calico comes with a rich library for exploring robots, called Myro. The Myro module allows students to control a real or simulated robot, take pictures, do image processing, make the robot speak, go through a maze, draw a picture, etc.

3.3 Graphics

This section describes Calico Graphics, a 2D graphics library for creating art, games, and animations in any of the Calico languages.

4. Conclusion

Acknowledgments

Acknowledgments, if needed.

References

[1] Da Vinci Machine for the JVM.
URL http://openjdk.java.net/projects/mlvm/pdf/LangNet20080128.pdf Accessed June 6, 2013.

[2] Project Lambda for the JVM.
URL http://openjdk.java.net/projects/lambda/ Accessed June 6, 2013.

[3] Project Jigsaw for the JVM.
URL http://openjdk.java.net/projects/jigsaw/ Accessed June 6, 2013.

[4] Project Nashorn for the JVM.
URL http://openjdk.java.net/projects/nashorn/ Accessed June 6, 2013.

[5] Dynamic Language Runtime. Microsoft.
URL http://en.wikipedia.org/wiki/Dynamic_Language_Runtime Accessed June 6, 2013.

[6] IronPython.
URL http://ironpython.codeplex.com/ Accessed on January 12, 2012.

[7] IronRuby.
URL http://en.wikipedia.org/wiki/IronRuby Accessed on June 6, 2013.

[8] GtkSharp.
URL http://www.mono-project.com/GtkSharp Accessed on January 12, 2012.

[9] Simple Direct Media Layer.
URL http://www.libsdl.org/ Accessed on June 6, 2013.

[10] C# SDL, wrapper for SDL.
URL http://cs-sdl.sourceforge.net/ Accessed June 6, 2013.

[11] Pete Shinners. Python Pygame Introduction.
URL http://www.pygame.org/docs/tut/intro/intro.html Accessed on June 6, 2013.

[12] IKVM Homepage.
URL http://www.ikvm.net/ Accessed June 6, 2013.

[13] Chaur Wu. 2010. *Pro DLR for .NET 4*. Apress, 2010.

[14] Calico Project.
URL http://calicoproject.org/ Accessed June 6, 2013.

[15] Douglas Blank, Jennifer S. Kay, James B. Marshall, Keith O'Hara, and Mark Russo. *Calico: A Multi-Programming-Language, Multi-Context Framework Designed for Computer Science Education*. SIGCSE12, February 29March 3, 2012, Raleigh, North Carolina, USA.

[16] Douglas Blank and Keith O'Hara. *The Calico Project for Robots in Education*. 2013. Under review.

[17] J. Rose. Lang Next.
URL http://cr.openjdk.java.net/ jrose/pres/201204-LangNext.pdf Accessed June 6, 2013.

[18] Squeak Virtual Machine.
URL http://www.squeak.org/Features/TheSqueakVM/ Accessed June 6, 2013.

[19] Comparison of application virtual machines.
URL http://en.wikipedia.org/wiki/Comparison_of_application_virtual_machines Accessed June 6, 2013.

[20] IronPython Performance.
URL https://ironpython.codeplex.com/wikipage?title=IP27A1VsCPy27Perf &referringTitle=IronPython%20Performance Accessed June 6, 2013.

[21] Ecma Standards.
URL http://www.mono-project.com/ECMA Accessed on January 12, 2012.

[22] Mono Virtual Machine and Runtime.
URL http://en.wikipedia.org/wiki/Mono_(software) Accessed June 6, 2013.

[23] Microsoft Community Promise. 2007.
URL http://www.microsoft.com/interop/cp/ Accessed June 6, 2013.

[24] eSpeak, text-to-speech.
URL http://espeak.sourceforge.net/ Accessed June 6, 2013.

[25] Bastian Steinert, Damien Cassou, and Robert Hirschfeld. *CoExist: Overcoming Aversion to Change*. DSL'12.

[26] Dynamic Language Runtime.
URL http://en.wikipedia.org/wiki/Dynamic_Language_Runtime Accessed June 6, 2013.

[27] Douglas Blank and Deepak Kumar. 2010. *Assessing the Impact of using Robots in Education, or: How We Learned to Stop Worrying and Love the Chaos*. 2010. AAAI Spring Symposium Series, Educational Robotics and Beyond: Design and Evaluation, Palo Alto, California, March 22, 2010 March 24, 2010.

[28] Tucker Balch, Jay Summet, Douglas Blank, Deepak Kumar, Mark Guzdial, Keith O'Hara, Dan Walker, Monica Sweat, Gaurav Gupta, Stewart Tansley, Jared Jackson, Mansi Gupta, Marwa Muhammad, Shikha Prashad, Natasha Eilbert, Ashley Gavin. 2008. *Designing Personal Robots for Education: Hardware, Software, and Curriculum*. Pervasive Computing, IEEE, volume 7, Issue 2, pages 5 - 9. April 2008.

[29] Douglas Blank. 2006. *Robots make computer science personal*. Communications of the ACM 49, 12, 25-27.

[30] PowerShell. URL http://en.wikipedia.org/wiki/PowerShell Accessed June 6, 2013.

[31] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, Jacob Matthews (editors). 2010. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press. April 2010.

[32] J. McCarthy, 1963. "A Basis for a Mathematical Theory of Computation", in P. Braffort and D. Hirshberg (Eds.), *Computer Programming and Formal Systems*, North-Holland.

[33] D. Friedman and M. Wand, 2008. *Essentials of Programming Languages, 3rd edition*. MIT Press.