Let Them Fail

Towards VM built-in behavior that falls back to the program

Tobias Pape
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
tobias.pape@hpi.uni-potsdam.de

Tim Felgentreff
Oracle Labs
Potsdam, Germany
tim.felgentreff@oracle.com

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

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

ABSTRACT

An important purpose of managed runtime environments like virtual machines is to provide *built-in behavior* to their programs, that is, behavior that cannot or may not be expressed in the program's language and is implemented inside the VM. For performance, stability, and security, such built-in behavior is typically rigorous, and if it deems its invocation to be erroneous, ill-suited, or just not quite right, it communicates this *failure* via exceptions at best. This implies that a VM's *sphere of influence* extends rather far into the realm of the program executed: at best, exceptions can be reacted to when built-in behavior fails, even though a program might be able to provide alternatives, improvements, or other ways of handling a failure.

By reflecting on how built-in behavior works in a Smalltalk system, we argue that the sphere of influence of programs on VMs is extensible with regard to built-in behavior. For that, we suggest to *let built-in behavior fail* and let programs decide whether to raise exceptions, try again, or simply do nothing at all.

CCS CONCEPTS

• **General and reference** → *Surveys and overviews*; • **Software and its engineering** → *Virtual machines*; *Runtime environments*.

KEYWORDS

Spheres of Influence, VMs, Built-in Behavior, Squeak/Smalltalk

ACM Reference Format:
Tobias Pape, Tim Felgentreff, Fabio Niephaus, and Robert Hirschfeld. 2019. Let Them Fail: Towards VM built-in behavior that falls back to the program.

Salon des Refusés, April 2, 2019, Genova, Italy

ACM ISBN 978-1-4503-6257-3/19/04…$15.00
https://doi.org/10.1145/3328433.3338056

In *Companion of the 3rd International Conference on Art, Science,and Engineering of Programming (Programming '19), April 1–4, 2019, Genova,Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3328433.3338056

1 HELICOPTER VMS

Virtual machines (VMs) are expected to be reliable executive agents of the program they are given.[1] In a manner of speaking, they tend to "hover over" the programs they are executing. With great care, and even more suspicion, they shall carry out a program's instructions, and quickly so. This view of VMs emerged from an environment where hosted language — such as Java, JavaScript, Python, Ruby, but also Smalltalk, etc. — take the abstraction barrier provided by the hosted-ness quite literally, relying on built-in behavior of the VMs without question.

An important effect of this bipartite division of responsibilities — oversimplified: programs do the sophisticated parts and VMs do the primitive parts — is that VMs have to make sure they get their part right. Every data passed, every behavior invoked has to be scrutinized thoroughly as to not break execution.

This is, in fact, a good thing: programmers gain trust in the platform, and VM, to get things right and be vocal about it. Yet, it is, in fact, a bad thing, too: sufficiently advanced programmers might increasingly see behavior that, while being reliable and sound, can be improved or adapted for unforeseen use cases, but they have no power to express themselves, as the VM would bail. Another aspect is, that scrutinizing precludes vagueness on principle. So even if the instruction to the VM is *almost* right, and to the human eye even understandable, the VM's duty is to complain.

But also the VM side might suffer. Introduction of new features, such as language concepts, that do not fit the way a language is currently used is often hard to do. A VM can maybe anticipate programs that are not yet apt to use a new feature, but programs that are aware of a new feature maybe have a hard time dealing with a VM not yet capable.

[1] We will use VM to collectively refer to managed runtime environments (MREs), VMs, and execution environments in general in this paper. Also, for the sake of simplicity we subsume libraries, frameworks, or applications running on a VM under *programs*.

⁂

Bailing and complaining typically means one of two things for a VM: crashing or throwing an exception. Leaving out the fatal variant, we are left with exceptions. However, everything unforeseen by the VM developers, everything deviating from a narrow definition of successful completion is subsumed under *exceptional* behavior, failure is nothing *normal*. Moreover, exceptions are challenging, too. VMs often prescribe the set of exceptions it would raise and the program has no say. Technically, exceptions can be painful to implement on VM level, especially when they have to be fast, which can happen when they are frequent, that is to say, not exceptional in the first place, but rather an alternative action. Admittedly, such alternative action typically starts out as merely conveying the fact of failure,[2] but seldom remains that way indefinitely.

We propose that VMs should share the responsibility they have with the program they are running — and their power accordingly; that is, to *allow the failure of built-in behavior being handled by the program* that induced the behavior. We propose to thereby reduce the VM's *sphere of influence* and increase that of the running program. We show a VM where this is indeed the case and argue that a VM behavior like this is beneficial to the development experience of several programming tasks.

2 BACKGROUND: SQUEAK AND ITS VM

Squeak [8] is a Smalltalk system derived from Smalltalk-80 [7]. Squeak's current default VM, the OpenSmalltalk VM [10], is based on the original Squeak VM [8] and applies additional optimization techniques to efficiently execute Smalltalk code.

The VM provides built-in behavior named *primitives* which are used to facilitate the communication between language and execution environment. Squeak primitives are invoked by first "tagging" methods with a primitive marker: `<primitive: aNumber>` or `<primitive: 'aName' module: 'aPluginName'>` . Whenever such a method is executed, the VM locates the primitive built-in behavior by its name or number, executes it, and if successful, returns a result. A method with a primitive tag can, however, include *fallback code*, which is executed whenever the execution of the primitive is not successful. The result of a primitive-tagged method hence is either the result of the built-in behavior or that of the fallback code.

The OpenSmalltalk VM is written in Slang, a subset of Smalltalk, and translated to C. This means that most primitives actually consist of executable Smalltalk code, which can come in handy when fallback code is necessary.

3 HOW SQUEAK FALLS BACK TO THE PROGRAM

We reflect on the fallback mechanism for Squeak's built-in behavior and show the broad applicability of fallback code. We identified nine variants of how fallback code is used, yet with slight overlap.[3]

3.1 Raise

For a large part of built-in behavior it is, in fact, a good idea to raise an exception to inform callers or end users in a structured way. In Squeak, the method `primitiveFailed` does exactly that. It is used whenever a primitive is unavailable or it encountered a situation it cannot handle (cf. Figure 1). This is the way probably most execution environments and VMs handle failure of built-in behavior, but instead of prescribing the reaction at the VM implementation level, Squeak is able to react at the language level.

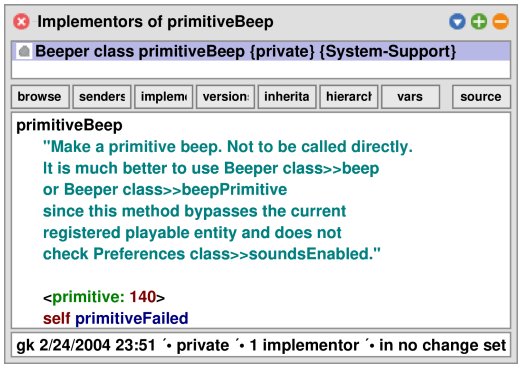


Figure 1: Raise an error when a primitive fails.

3.2 Ignore

Complement to explicitly raising an exception on failing built-in behavior, Squeak's fallbacks support ignoring failing behavior. That is, in case a primitive fails or is absent, fallback code can simply be left out (cf. Figure 2). The effect is equivalent to any other empty Smalltalk method: in case of failure, the receiver (viz. `self`) is returned.

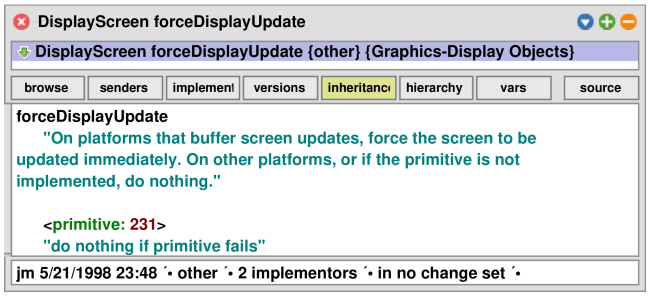
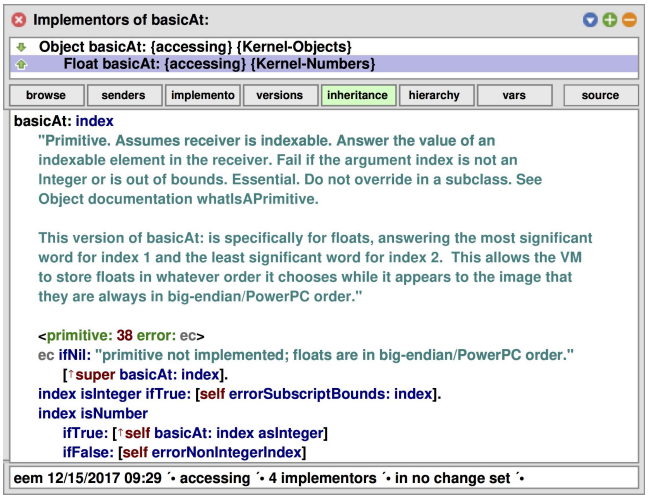


Figure 2: Ignore a failing primitive.

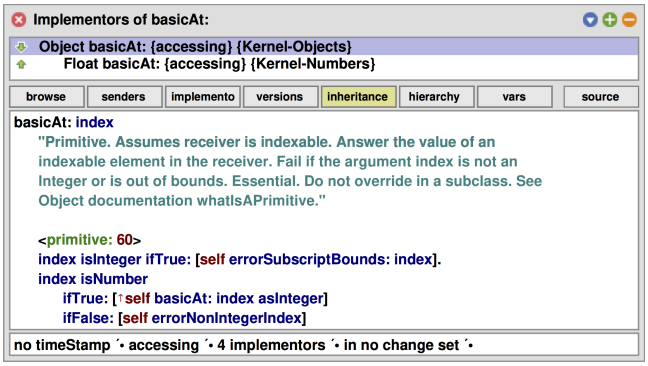
3.3 Chain

Fallback code can be put to good use to also keep inheritance and polymorphism working correctly. For example, in Squeak, some objects may want to provide specialized behavior for certain access patterns — such as accessing the high and low word in a double word float object (cf. Figure 3a). However, if the primitive access to those fails, the inherited built-in behavior for accessing an object's memory can be re-used using an object-oriented super call in the Smalltalk code (cf. Figure 3b), rather than some special handling for the primitive.

[2] That is why we will call it *failing* here.
[3] The variants marked † and ‡ are only more broadly applicable to metacirular VMs but included to show the diverse ways of utilizing fallback code.



(a) A primitive might fail and resort to its inherited behavior (via `super`-send).



(b) The inherited behavior is itself built-in.

Figure 3: Invoking a different primitive when one fails.

This way, the benefits of an object-oriented system are not lost only because multiple behaviors are implemented as primitive and neither are wrapper functions necessary to chain the primitives explicitly.

3.4 Adapt

Since fallback code in Squeak is ordinary Smalltalk code, it is possible react to failure with adaptation to the current situation. For example, to instantiate (or construct) an object in Smalltalk, the message `new` is sent to a class. This is actually an idiom and not part of the language *per se*. Typically. `new` is implemented in a way that it calls `basicNew`, which invokes the built-in behavior (cf. Figure 4). However, there is a certain category of objects, such as Arrays, that require the number of elements to be given at the time of creation and, hence, requires sending `basicNew:` instead. Squeak copes for code not completely complying with this idiom. That is, when a *variable* object like an Array is instantiated via `basicNew`, the primitive fails, and the fallback code calls *another* primitive with a default number of elements (zero, in this case).

That way, it is possible to use primitives in *almost* the right way and adapt to such situations *post hoc*.

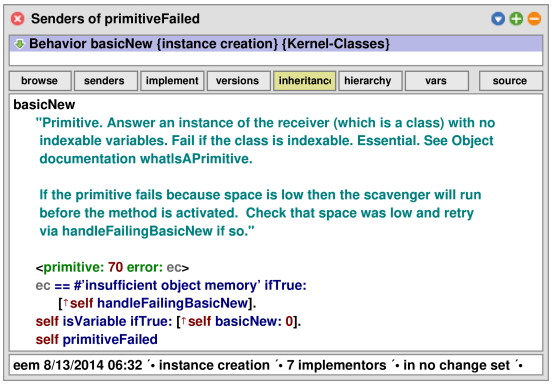


Figure 4: Handle a nearly-correct primitive usage.

Nota bene: The code shown (Figure 4) actually includes additional strategies to deal with failure: first, it reacts to a certain *kind* of failure (communicated by the VM as a symbol or string via the variable `ec`) by "chaining" to a different primitive, and in case that or the adaptation here does not work, "raises".

Another way of adapting to failing behavior is returning defaults upon failure. This idiom is also common in Squeak fallback code.

3.5 Retry

Built-in behavior, regardless of how it implemented, by its very nature only has access to a subset of the capabilities of the implemented language. This also holds for Squeak and its subset Slang (see Section 2). Thus it can be a lot of effort to support a wide range of inputs in primitives. An example for this can be found in the *BitBlt* primitive that is used to combine rectangular areas of one display form with another according to some combination rule (cf. Figure 5). This primitive takes a large number of arguments to specify the source and target forms, the rectangular areas in question, and color palette conversions. It would be a lot of work to support all kinds of parameters in the primitive code, so the primitive supports only integers and uncompressed forms as parameters. If an argument is given that is of a different type, the primitive fails.

The fallback now attempts to coerce all arguments into appropriate types by sending messages such as `asInteger` to round non-integer values or `unbent` to decompress display forms. Afterwards, the same primitive is simply re-tried.

The advantage here is that programmers can easily add new abstractions in the language for numbers of display forms without having to touch the primitive code; as long as coercion to the accepted types is possible.

3.6 Rescue

The reasons for built-in behavior failing to do what was intended are not always clear. Sometimes it is undesirable to either just raise or ignore the failure, but the situation is not as clear cut to simply chain to another primitive or adapt.

In this case, the fallback code in Squeak can be used to actually carry out the functionality that the primitive was intended to

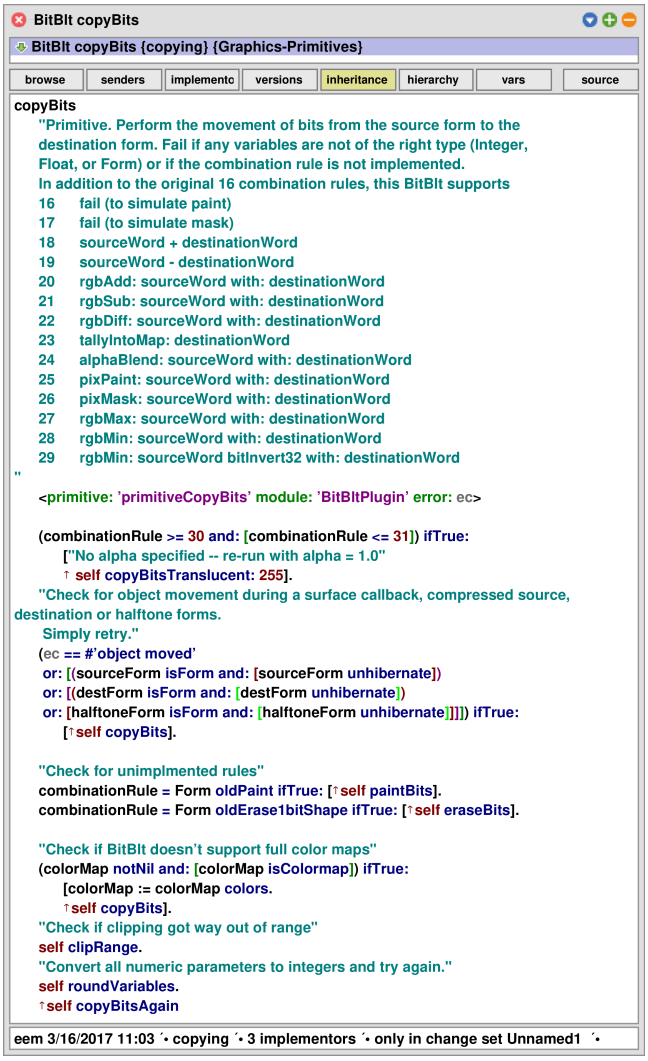


Figure 5: On failure, adjust BitBlt's object state and retry the primitive.

provide. In the example (cf. Figure 6), a fast Fourier transform is computed, either via primitive or using Smalltalk code. In fact, for several methods in Squeak, this variant is used not to react to failure of built-in behavior, but rather the other way round: the primitive is *optionally* available to speed up certain computation that could be too slow when executed as Smalltalk.[4]

Note, however, that now manual intervention is necessary to keep the primitive implementation and the fallback code in sync.

3.7 Forgo

Squeak's primitive fallback behavior can also be used to introduce new concepts in a backwards-compatible way. Figure 7, for example, shows a method for instantiating an immutable `Cons` cell. Using

[4] A more in-depth discussion of such *algorithmic primitives* is available in [6].

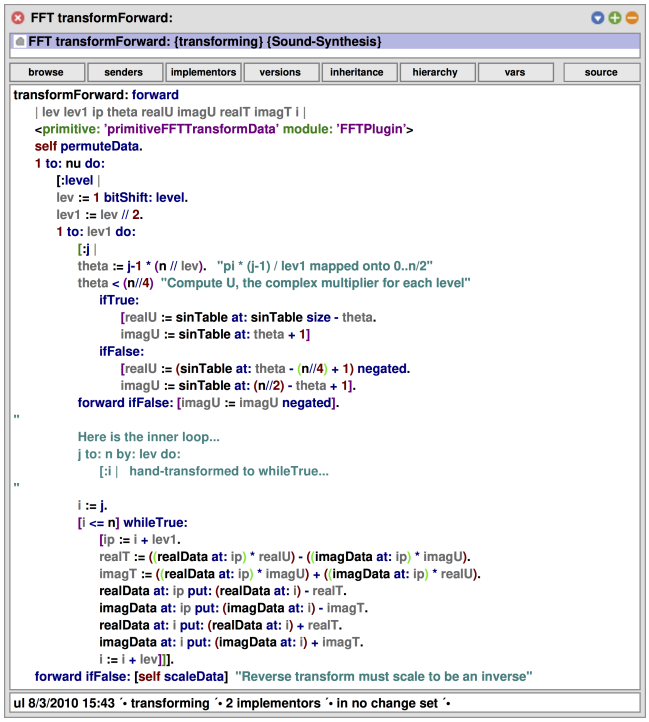


Figure 6: Fall back to non–built-in behavior on failure.

fallback code, the behavior can be mimicked in case the underlying execution environment is unable to guarantee immutability properties. A VM without the `ImmutabilityPlugin` would happily allow these objects to be mutated which can simply be avoided by overriding appropriate methods in `Cons`, so that an error is raised instead.

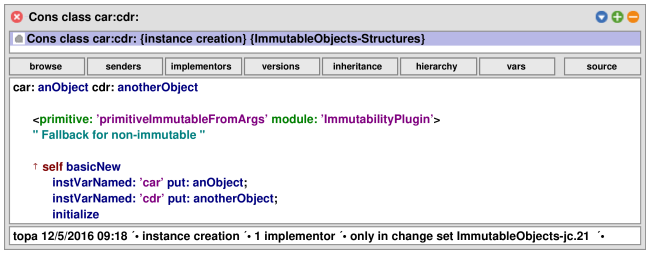


Figure 7: Fallback code can be used to mimic certain behavior.

3.8 Reuse†

The Squeak VM is predominantly written in Slang, a Smalltalk subset, and so are numerous primitives. The effect is that such primitives can both be executed as Smalltalk code or translated to C for compilation into the VM. This makes it possible for certain primitives to be their own fallback code (cf. Figure 8). While this approach has its limitations (for example, no access to certain VM internal interfaces), it avoids any behavioral deviation between primitive and fallback code.

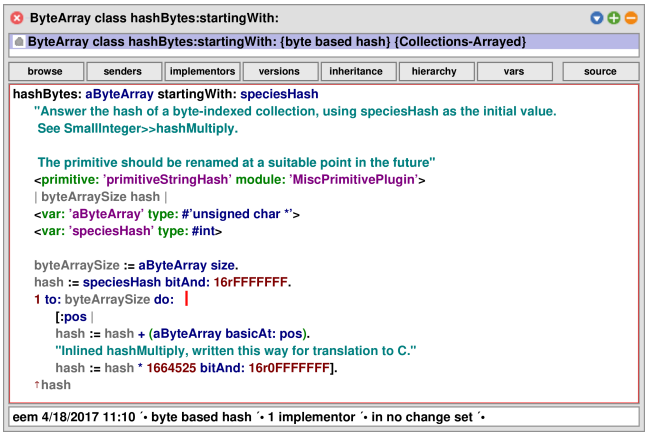


Figure 8: Fallback code that also serves as source for translation to C.

3.9 Simulate‡

A more advanced version of handling the absence of certain built-in behavior — as a special case of failing — is provided with the R/SqueakVM [2, 5] implementation of Squeak's VM. When it detects that the invocation of a method requests a primitive not provided by the VM, a configured callback can be sent to the send's receiver (or its class, for that) matter. The callback is called `simulatePrimitive:args:`, as it makes it possible to re-use a so-called *simulation* version of the primitive's implementation [6]. This is possible, as the development of the OpenSmalltalk VM happens, in fact, within Squeak itself; the complete virtual machine can be executed from *within* a running Squeak instance, then called the *Simulator*. Due to machine restrictions, the translated-to-C code and the run-in-Simulator code sometimes differ slightly. Compared with normal fallback code, the distinctive feature of Simulator code is that it is written as if it is run *inside* the VM, and hence using different interfaces and behavior.

This variant of fallback code is useful when the built-in behavior already has fallback code to handle the "nearly-right" case of failing primitives. In Squeak, the low-level graphics routine, BitBlt's `copyBits` is written that way, but cannot deal with the absence of the primitive. R/SqueakVM can however omit the (rather complex) primitive and relies on the code of the BitBltSimulator, providing the simulator code for BitBlt.

Note that this variant of fallback behavior needs explicit simulation code which might be provided by a meta-circular VM.

Simulate (Sometimes). A variation of the simulation fallback approach has been tried out in GraalSqueak [11]. Instead of fully simulating a primitive, such as BitBlt's `copyBits`, the VM only falls back to the simulation code for certain arguments. It only *partially* implements that primitive. Since GraalSqueak is implemented in Truffle, the language implementation for the GraalVM, it is possible to implement only the most performance-critical code paths on the VM-level by providing specializations based on certain argument values.

In the case of `copyBits`, for example, specializations could be activated for the most complex combination rules that specify how BitBlt combines two bitmaps into one. For all other code paths that are not covered by such specializations, the VM lets the primitive fail and falls back to "simulate".

⁂

The ways to handle failing built-in behavior in Squeak are manifold. However, they are *only* because the VM relinquishes parts of its sphere of influence to the running program. In the case of Squeak, programs thus gain a lot of flexibility and room to improve on failing built-in behavior.

4 DISCUSSION

4.1 Failure is not exceptional

While it could be dismissed as mere naming issue, mix-matching failure-handling[5] and exceptions leads to a dilemma: either failure is truly the exception — then even trying to open a file without proper access rights may be hard to express — or it is common an hence not exceptional.

That being said, we cloud treat exceptions just as means to convey the information that built-in behavior failed. However, a built-in that raises puts the — possibly plenty of — *callers* of that built-in in charge of handling that exception; if not, exceptions uncatched typically end up effecting the other type of VMs bailing, that is, they crash.

The way fallback code works in Squeak, developers are very much encouraged, if not forced, to deal with primitives failing at its root, that is the methods with the primitive tag, *not* their callers. We consider this a kind of psychological advantage. This could be approximated in other environments by wrapping all built-in behavior with exception-catching functions and only calling the latter.

4.2 Spheres of influence challenged

When a program has the power to react to failing built-in behavior in an arbitrary way, it certainly has the responsibility to do so, too. Yet, it can be onerous to do the right thing, because the right thing might be complex, slightly different, or simply not known.

Too simple fallback code could cover up actually erratic VM behavior. Without a fine-grained set of messages between VM and program, separating recoverable or handleable failure from irrevocable errors becomes guesswork. In fact, the possibility to hand an error code from the failed primitive to the fallback code in Squeak serves exactly this purpose and — as later addition not present in Smalltalk-80 — may be actually a reaction to this discovery.

Whenever a fallback's intention is to "rescue", "reuse", or "simulate", there is a chance that the observed behavior of primitive and fallback get out of sync. Since typically VMs change less often that the programs they run, care must be taken on the program level to synchronize behavior in such cases.

As with all powerful tools, a program with greater influence on what it does when primitives fail poses an educational challenge. Programmers have to be aware of what they *can* do and what they *should* do and what not. Squeak partly anticipates that.

[5] Or alternative action, for that matter.

A lot of primitives bear a comment referring to a method `Object whatIsAPrimitive`.[6] This methods includes a quite extensive introduction to the concept of primitives and how fallback code is to be read and written, with examples. This could serve as starting point for similar resources to make programmers ready for powerful fallback code.

5 RELATED WORK

Most MRE- or VM-based programming languages provides means to invoke built-in behavior different from a mere foreign function interface (FFI): Java Native Interface (JNI) [9] for Java, Python C extensions [4, 3], or Ruby Native Extensions [12], to name just a few. None of them provide the means to execute fallback code as presented here.

However, reacting to the absence of built-in behavior is not completely uncommon. The following pattern is rather common in Python:

```
try:
   import cPickle as pickle
except:
   import pickle
```

This code tries to load a C-written module (`cPickle`) using the name of the Python-written module (`pickle`). In case this fails, the actual, Python-written module is loaded. This is typically used when the C-written module is merely a performance-improved, interface-compatible version of a Python module, possibly with reduced extensibility. We would classify this under "rescue", but on a module granularity instead of method/function granularity. Similarly, it is possible to write wrappers for built-in functions, as in Listing 1. However, this depends on the built-in behavior actively using Python's exception mechanism and seems to be not very common.

Listing 1: Wrapping a built-in Python function and providing custom exception handling.

```
type(sum)
# => <type 'builtin_function_or_method'>
builtin_sum = sum

def sum(sequence, start=0):
    try:
        return builtin_sum(sequence, start)
    except Exception as e:
        print "NO: %s: %s" % (type(e).__name__, e)

type(sum)
# => <type 'function'>
sum([2, 3, 4], object())
# => "NO: TypeError: unsupported operand type(s) for +: 'object' and 'int'"
```

In Ruby, this works likewise, except that `require` there is a function and not syntax. Others, mostly dynamic and scripting languages such as Lua or R, behave comparably.

In contrast, Rubinius,[7] a Ruby implementation, inherited (among other things) the Smalltalk idea of a primitive method with fallback code similar to what we presented.

A completely different approach is taken by Erlang [1]: rather than *handling* failure, the whole environment is tuned to tolerate failure or crash and will deal with it by restarting affected processes. However, handling things that are not actually failures is hence not as easy to achieve.

[6] The method's content is available on the web: https://marianopeck.wordpress.com/2011/06/03/primitives-pragmas-literals-and-their-relation-to-compiledmethods/ (last accessed 2019-01-20).

[7] https://rubinius.com/ (last accessed 2019-01-20).

6 CONCLUSION

While common, it is not necessary for a VM to answer the failure of built-in behavior solely with exceptions. With a bit of courage, it can relinquish parts of its sphere of influence and let programs decide what to do in such a case. The case of Squeak has shown that — in a manner of speaking — "grown up" programs can handle failure, in both responsible and creative ways, even facilitating the adoption of new language or VM features. We think this would make a good evolutionary step for VMs of other languages.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of Oracle Labs, the HPI Research School, and the Hasso Plattner Design Thinking Research Program.

REFERENCES

[1] Joe Armstrong, Robert Virding, Mike Williams, and Claes Wikstrom. 1996. *Concurrent Programming in Erlang*. (2nd edition). Prentice Hall, (January 16, 1996). ISBN: 978-0-13-508301-7.

[2] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. 2008. Back to the future in one week — implementing a Smalltalk VM in PyPy. In *Self-Sustaining Systems* (Lecture Notes in Computer Science). Robert Hirschfeld and Kim Rose, editors. Volume 5146. Springer Berlin Heidelberg, 123–139. ISBN: 978-3-540-89275-5. DOI: 10.1007/978-3-540-89275-5_7.

[3] The Python Community. 2016. *Extending and Embedding the Python Interpreter*. The Python Community. (July 26, 2016). Chapter 4. Building C and C++ Extensions. Retrieved 2019-01-20 from https://docs.python.org/3/extending/building.html.

[4] The Python Community. 2017. *Extending and Embedding the Python Interpreter*. The Python Community. (November 24, 2017). Chapter 1. Extending Python with C or C++. Retrieved 2019-01-20 from https://docs.python.org/2/extending/extending.html.

[5] Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. 2016. 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) Article 21. ACM, Prague, Czech Republic, 21:1–21:10. ISBN: 978-1-4503-4524-8. DOI: 10.1145/2991041.2991062.

[6] Tim Felgentreff, Tobias Pape, Lars Wassermann, Robert Hirschfeld, and Carl Friedrich Bolz. 2015. Towards reducing the need for algorithmic primitives in dynamic language vms through a tracing jit. In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (ICOOOLPS '15) Article

7. ACM, Prague, Czech Republic, 7:1–7:10. ISBN: 978-1-4503-3657-4. DOI: 10.1145/2843915.2843924.

[7] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. The Blue Book. Addison-Wesley Longman, Boston, MA, USA. ISBN: 978-0-201-11371-6.

[8] Daniel H. H. Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, (October 1997), 318–326. DOI: 10.1145/263698.263754.

[9] Sheng Liang. 1999. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN: 0-201-32577-2.

[10] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. 2018. Two Decades of Smalltalk VM Development: Live VM Development Through Simulation Tools. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (VMIL 2018). ACM, Boston, MA, USA, (November 4, 2018), 57–66. ISBN: 978-1-4503-6071-5. DOI: 10.1145/3281287.3281295.

[11] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2018. GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (ICOOOLPS '18). ACM, Amsterdam, Netherlands, 30–35. ISBN: 978-1-4503-5804-0. DOI: 10.1145/3242947.3242948.

[12] Dave Thomas. 2010. *Extending Ruby 1.9: Writing Extensions in C*. The Pragmatic Bookshelf, Raleigh, North Carolina. http://media.pragprog.com/titles/ruby3/ext_ruby.pdf.