The operating system: should there be one?

Stephen Kell
Oracle Labs
srkell@acm.org

Abstract

Operating systems and programming languages are often informally evaluated on their conduciveness towards composition. We revisit Dan Ingalls' Smalltalk-inspired position that "an operating system is a collection of things that don't fit inside a language; there shouldn't be one", discussing what it means, why it appears not to have materialised, and how we might work towards the same effect in the postmodern reality of today's systems. We argue that the trajectory of the "file" abstraction through Unix and Plan 9 culminates in a Smalltalk-style object, with other filesystem calls as a primitive metasystem. Meanwhile, the key features of Smalltalk have many analogues in the fragmented world of Unix programming (including techniques at the library, file and socket level). Based on the themes of unifying OS- and language-level mechanisms, and increasing the expressiveness of the meta-system, we identify some evolutionary approaches to a postmodern realisation of Ingalls' vision, arguing that an operating system is still necessary after all.

Categories and Subject Descriptors D.1.5 [*Programming techniques*]: Object-oriented programming; D.4.m [*Operating systems*]: Miscellaneous

General Terms Languages, design

Keywords Unix, Smalltalk, Plan 9, metasystem, composition, binding, integration

1. Introduction

Writing in the August 1981 "Smalltalk" issue of Byte Magazine, Dan Ingalls set forth various design principles behind the Smalltalk language and runtime [Goldberg and Robson 1983], and addressed the issue of integration with the operating system as follows [Ingalls 1981].

> An operating system is a collection of things that don't fit into a language. There shouldn't be one.

Although not stated explicitly, we can infer that Ingalls' vision for there "not being" an operating system would include gradually pulling more and more system functionality (e.g. isolated processes, filesystems, network stacks) into the Smalltalk runtime, where it could be exposed in the form of higher-level abstractions (e.g. as persistent and remote objects) rather than the byte-streams and raw memory interfaces of Unix—which seems unquestionably to be one system to which his "very primitive" refers.

It appears that this change has not happened—at least not yet. Was there a real benefit in whole-system design underlying Ingalls' position? If so, is it achievable? If so, would there remain any need for a programmer- or user-facing operating system? In this paper, we make a case for answering all these questions in the affirmative, consisting of the following contributions.

- We identify the potential benefits of Ingalls' vision, and contrast these with parallel developments in Unix relating broadly to the concerns of composition.
- We argue that the natural trajectory of the Unix design, extrapolated through Plan 9 and beyond, yields an object abstraction mostly equivalent to that of Smalltalk.
- We map a large number of composition "point fixes" in Unix systems to features of a Smalltalk-like environment, each of which they are replicating for some set of use cases.
- We argue that this "lurking Smalltalk" could be exploited to bring about many aspects of Ingalls' vision in an evolutionary fashion, and sketch several approaches to this.

2. The Smalltalk wishlist

In saying that there "shouldn't be" an operating system, what benefits is Ingalls seeking? Clearly, the problem being addressed is that of *complexity* in software (the same article emphasises "management of of complexity"), and that Smalltalk's approach is to provide well-designed abstractions which are *compositional* (which we take to be the essence of any programming language). Although the article does not list the intended benefits explicitly, we can infer that the following general benefits are probably included.

Programmatic availability The Smalltalk programming abstraction is also available to system-level tasks. Programmers can to write code "in the same way" against both user-defined and system-defined abstractions (e.g. processes, devices), also allowing the application of existing Smalltalk code (say, the famous Collections library) to these new target domains. For example, maintaining a configuration file, generating a coredump or mounting a filesystem all cease to require mechanism-specific code (disk–memory marshalling, object file manipulations, invoking the mount system call); they are simply rendered as (respectively) accessing a (persistent) configuration object, cloning a process object (likely stopping and persisting the copy), or pushing a new object into some delegation chain. The late-bound semantics and interactive interface offered by Smalltalk allow it to subsume both programming and "scripting" (as offered by the Unix shell).

Descriptive availability The pervasive metasystem of Smalltalk enables cheap provision of "added value" services expressible at the meta-level, such as human-readability, visualisation, interactive data editing, debugging, or data persistence. Extending the reach of

http://dx.doi.org/10.1145/2525528.2525534
PLOS'13, November 3 2013, Farmington, PA, USA

this meta-level to system-level state would amplify these benefits (when inspecting device state, debugging device drivers, persisting device configuration, and so on).

Interposable bindings The late-bound, message-based interfaces of objects provide strong interposability properties: clients remain oblivious of the specific implementation they are talking to. In turn, this simplifies the customisation, extension or replacement of parts of a system, all of which can be rendered as interposition of a different object on the same client.[1] The concept of interposition presupposes a mechanism by which references to objects are acquired and transmitted. This process is *binding*. In Smalltalk there is one general mechanism for object binding, which is the flow of object references in messages. Binding is also prominent in Unix's design, as we will contrast shortly.

Although these concerns are integral to Smalltalk, they are not foreign to operating system designers either, whose work is often evaluated on its conduciveness towards composition. We next consider Unix and its successor Plan 9 from the perspective of these concerns.

3. Unix and Plan 9: the tick-list

We consider firstly the 5th edition Unix described by Ritchie and Thompson [1974], then continue to later Unices and Plan 9.

Programmability Ritchie and Thompson wrote that "since we are programmers, we naturally designed the system to make it easy to write, test, and run programs". Indeed, Unix exposes multiple programmable interfaces: the host instruction set (a large subset of which is exposed to the user via time-sharing processes created from a.out images), the various system calls (which embed into the host instruction set, extending it with operating system services), the shell (which abstracts the same interfaces in a manner convenient for interactive and scripting-style use) and the C language. These four means cohere to some extent. The last of the four, C, is an abstract version of the first (both concerning in-process "application" programming). Meanwhile, the shell can be considered an abstract version of the system call API, since it specialises in file- and process-level operations. We call the latter kind of programming "file-or-device" or just "device" programming. This remaining twofold distinction runs deep: between application mechanisms (which, aside from trapping into system calls, are opaque to the operating system) and file mechanisms (which are the operating system's reason for being). We will call this the *application–device split*.

Description Unix was original in exposing diverse objects—program binaries, user files, and devices—in the same namespace, in a unified way. This includes names and other metadata, along with enumerable directory structures. Although primitive, this is clearly a meta-system. For instance, enumeration of files in a directory corresponds closely to enumeration of slots in an object, as expressible using the Smalltalk meta-object protocol. However, Unix's meta-system is selective in coverage and content—the system predetermines what state is exposed to the filesystem, the metadata and operations are somewhat specialised for storage systems (sizes, timestamps, etc.), and the facility for exposing state at this meta-level is not extended to application code. While subsequent developments have integrated additional operating system state into the filesystem model, including processes [Killian 1984] and device state (as with Linux's sysfs [Mochel 2005]), they have not changed this basic property.

Interposable bindings Thompson and Ritchie stated as a goal for Unix the property that "all programs should be usable with any file or device as input or output". This is a clearly an interposability property. It was successfully achieved by unifying devices with files—the famous "everything is a file" design. Note, however, its tacit characterisation of applications as having *unique* input and output streams. The streams stdin and stdout are easily substitutable: they exist in every process, and the parent can bind them (using dup()) to any file or device it can open. However, many other cases of interposition are not supported.[2]

One example is how user code cannot quite "be a file", because only files may be opened by name. (By contrast, for programs using only parent-supplied file descriptors, pipe() serves for this purpose.) The same property means that programs accessing specific files or devices may only be redirected to *user-selected* files if the developers had the foresight to accept the file name as a parameter. Sometimes this foresight is lacking (as known to anyone who has resorted to recompiling a program just to replace a string like "/dev/dsp"). In Smalltalk this foresight is not necessary, because this kind of definitive early binding is not possible.

Contrasts "Late binding everywhere" is one property which helps Smalltalk ensure interposable bindings, and which on Unix is left for the user to implement (or not). We can note several other contrasts. While the Unix filesystem is a primitive metasystem, it lacks any notion of user-defined "classes", which in Smalltalk exists to describe commonalities between between both user- and language-defined abstractions. In Unix, explicit classes are unnecessary, since objects in the filesystem are always of one of three (implicit) classes (files, directories, or devices; later, symbolic links, named pipes and sockets would be added to this list). Meanwhile, user-defined classes need not be supported because Unix remains pointedly oblivious to user code.

Another way of looking at this is that the operating system concerns itself with *large objects* only, where we crudely characterise files as large objects, in contrast to the units of data the size of program variables, such as allocated on the process stack or by malloc(), which are generally much smaller. The specification of the mmap() system call in 4.2BSD[3] and the advent of unified virtual memory systems [Gingell et al. 1987] would cement a unification of files and memory objects, but *only* for the case of large objects. This was primarily since their interfaces work at page-sized granularity, being neither convenient nor efficient for smaller objects. (Of course, Unix filesystems certainly support the case of small files. "Large objects" is therefore our shorthand for "objects selected by the programmer to be managed as mapped files"—likely for their large size, but perhaps also to enable their access via inter-process communication, as with the example of small synthetic files in the /proc filesystem.)

A consequence of offering only these large-object abstractions is that Unix is tolerant to diversity in how smaller objects are managed. Unix processes happily "accommodate" diverse implementations of language-level abstractions, albeit in the weakest possible sense: by being oblivious to them. By remaining agnostic to application-level mechanisms (in the form of programming languages and user-code libraries), Unix likely boosted its own

[1] Unix often talks about *redirection* instead of interposition; we consider these synonymous. ("Redirection" sounds slightly stronger, but consider that there is no obligation for an interposing object to make any use of the implied interposed-on object.)

[2] We note that the difficulty sticking to parent-bound I/O streams is not that only input and output streams are supported, since a parent process may dup() arbitrarily many descriptors before forking a child. Rather, it is that the set of streams must be enumerable by the parent in advance. This precludes cases where the eventual number or selection of I/O streams depends on program input.

[3] Although specified in the 4.2BSD design, around 1982, and described in the Programmer's Manual of the 4.3 release in 1986, this interface would remain unimplemented in any BSD release until 1990's 4.3BSD-Reno.

longevity, but at a cost of *fragmentation*. This included not only fragmentation of system- from user-level mechanisms, but also fragmentation *among* system-level mechanisms (noting the various binding mechanisms we have identified), and finally, fragmentation within opaque user-level code (noting that each language implementation typically invents its own mechanisms for object binding and identity, a.k.a. conventions for representing and storing object addresses). The result of this fragmentation—one which has only grown since Ingalls' article—is an extent of noncompositionality which is anathema to the "unified" ideal (held by both Smalltalk and, initially, Unix). It has the effect of ensuring that different software ecosystems are kept separate, and that logically sensible compositions are difficult or impossible to achieve. If diverse binding mechanisms were not enough fragmentation, the addition of independently developed protocols and data representations "in the small" adds even further impediment to composition.

We should counter, however, that Smalltalk itself has no solution to fragmentation. Its solution is "don't fragment; use Smalltalk for everything!". This sweeping position is what Noble and Biddle [2002] call a modernist "grand narrative". By contrast, Unix succeeds in existing in the postmodern reality of diverse, independently developed, mutually incoherent language- and application-level abstractions, by virtue of its obliviousness to them.

4. Post-Unix trajectories: Plan 9 and beyond

The need for greater unification in Unix is well known. Since its initial design, a trend in Unix has been to unify around the filesystem abstraction, by opening it up to new and diverse uses. As noted previously, exposing processes as files [Killian 1984] created a cleaner and faster alternative interface to process debugging and process enumeration. VFS [Kleiman 1986], a kernel-side extension interface for defining new filesystems, later became a central feature of all modern Unix implementations. Plan 9, Bell Labs' spiritual successor to Unix, embraces the filesystem to an unprecedented extent. Its design, pithily stated, is that "everything is a [file] server"—a system is a (distributed) collection of processes serving and consuming files, or things superficially like them, using a standard protocol (9P) that is transport-agnostic. Applications serve their own filesystems, and essentially all inter-process functionality is exposed in this fashion. To illustrate the design of Plan 9 and its conducivity to composition, Pike recently recounted[4] the following impressive anecdote about the design's properties.

> A system could import… a TCP stack to a computer that didn't have TCP or even Ethernet, and over that network connect to a machine with a different CPU architecture, import its /proc tree, and run a local debugger to do breakpoint debugging of the remote process. This sort of operation was workaday on Plan 9, nothing special at all. The ability to do such things fell out of the design.

The expanded use of files and servers allowed several simplifications relative to the Unix syscall interface. For example, gone are ioctl() and other device manipulations process operations such as setuid() or nice() and the host of Berkeley sockets calls (which added yet another naming and binding mechanism to Unix). Replacing them are a generalised binding mechanism—essentially bind() by the server and open() by the client—and simple reads and writes to files, including on a selection of *control files*. These are files with arbitrary request-response semantics: a client writes a message, and from which then reads back a response. Any operation can be expressed in this way; indeed, it is not-so-uncannily reminiscent of message-passing in Smalltalk.

As the filesystem's use has expanded, its semantics have become less clear. What do the timestamps on a process represent? What about the size of a control file? Is a directory tree always finite in depth (hence recursable-down) or in breadth (hence readdir()-iterable)? Although some diversity was present even when limited to files and devices (is a file seekable? what ioctls[5] does the device support?), semantic diversity inevitably strains a fixed abstraction. The result is a sytem in which the likelihood of a client's idea of "file" being different from the file server's idea is ever-greater. It becomes ill-defined whether "the usual things" one can do with files will work. Can I use cp to take a snapshot of a process tree? It is hard to tell. The selection of what files to compose with what programs (and fixing up any differences in expected and provided behaviour) becomes a task for a very careful user. Unlike in Smalltalk, semantic diversity is not accompanied with any meta-level descriptive facility analogous to classes.

For the impressive compositionality of his anecdote, Pike credits the filesystem abstraction of Plan 9, i.e. the property that "all system data items implemented exactly the same interface, a file system API defined by 14 methods". (Given the few semantics which are guaranteed to be ascribed to a file, 14 seems a rather large number.) Reading more closely, a different property of Plan 9—the the network transparency of server access—is at least jointly responsible. It is no coincidence that Smalltalk objects, like Plan 9 files, are naturally amenable to a distributed implementation [Schelvis and Bledoeg 1988] and that Alan Kay has recollects how from a very early stage he "thought of objects being like biological cells and/or individual computers on a network" [6][7]

Proposals for applying Plan 9's file-server abstraction still further are easy to find. One example is shared libraries: Narayanan blogged[8] a sketch of a proposal for shared file servers replacing shared libraries, using control files to negotiate a precise interface version. In both this case and Pike's quotation above, what is actually being articulated is the desire for three properties which, of course, Smalltalk already has: a network-transparent object abstraction (an unstated enabler of Pike's composition scenario), a metasystem (bundled into the unifying API Pike mentions) and late binding (for addressing the versioning difficulties mentioned by Narayanan).

It now seems reasonable to declare "file" (in the Plan 9 sense) and "object" (in the Smalltalk sense) as synonymous. Both are equally universal and more-or-less deliberately semantics-free. However, still distinguishing Smalltalk from Plan 9 is the former's meta-system and inclusiveness towards objects large and small. Whereas Plan 9 applications which must implement a 14-method protocol to reify their state as objects, Smalltalk's objects have this "by default". Moreover, the notion of classes allows, at the very least, some semantic description of an object.

Before continuing, it is worth noting that around the same time as Plan 9, research into microkernels and vertically-structured operating systems (or "library OSes") brought new consideration of binding and composition in operating system designs [Bershad et al. 1995; Engler and Kaashoek 1995; Leslie et al. 1996; Rashid et al. 1989]. These systems were mostly designed with a somewhat object-oriented flavour. Indeed, a key consideration was how to replicate a largely Smalltalk-like object- or messaging-based abstraction in the presence of the fine-grained protection

[4] in his 2012 SPLASH keynote; slides retrieved from http://talks.golang.org/2012/splash.article on 2013/7/20

[5] ioctl() first appeared in 7th Edition Unix, although calls including gtty and stty are its forebears in earlier versions.

[6] Various sources on the web attribute this statement to Kay.

[7] Indeed, a Smalltalk-style notion of "object" corresponds closely to the notion of "entity" in the OSI model of networking [Zimmermann 1988].

[8] at http://kix.in/2008/06/19/an-alternative-to-shared-libraries/, retrieved on 2013/7/20

boundaries—and moreover, how do so with high performance. In at least one case, a dynamic interpreted programming environment was developed atop the core operating system, furthering this similarity [Roscoe 1995]. These systems' results are encouraging testament to the feasibility of acceptable performance in a system of fine-grained protection domains. More recently, Singularity [Hunt and Larus 2007] is arguably a culmination of work on this issue, offering the radical solution of avoiding hardware fault isolation entirely and relying instead on type-based software verification. Like Smalltalk, however, these systems offer only a grand narrative on how software could and should be structured. Unlike Smalltalk, their programming abstractions were something of a secondary concern, lacking a true aspiration to influence the fabric and construction of user-level software. Accordingly, they have been the subject of substantially less application programming experience. For our purposes, protection and performance are both orthogonal concerns, so further consideration of these systems would add little content to our discussion.

We focus instead on how to approach the fragmented collection of Unix composition mechanisms and evolutionarily recover some of the benefits offered by the unified ideal of Smalltalk.

5. The Lurking Smalltalk

It turns out, perhaps surprisingly, that the Smalltalk-style facilities we just identified in Plan 9—a generic object abstraction, a metasystem (albeit primitive), and interposable late binding—are present in abundance in modern Unices too. However, they are to be found in Unix's characteristic fragmented form. Countless Unix implementations of languages, libraries and tools have grown mechanisms and/or recipes catering to each of these requirements. We survey them here, arguing their existence is the sign of a "lurking Smalltalk". Unfortunately, their fragmented nature renders them usable only by experts solving specific particular use cases—rather than with the naturalness and immediacy of a *designed* system. We will consider how to exploit them more effectively in the following section.

5.1 Lurking programmability

Programmability is abundant in Unix ecosystems, but often in awkward-to-use forms. Aside from the shell, the C compiler and whatever other language implementations are available, many applications implement their own configuration language or other "mini-language". Why are these mini-languages necessary? Sometimes they are a domain-specific form optimised for the domain at hand. But in others, they are simply an expedient form of exposing "good enough" configurability or customisability. Some straddle the line: for example, tcpdump's packet predicate language is concise for experts, but complex and quirky for newcomers. There is a case for saying it should be possible (but not mandatory) to write tcpdump predicates in a language of the user's choice. Similarly, administrators' jobs would often be easier if they could write configuration logic in a language of their choosing, rather than an idiosyncratic config file format.

(This is a strong requirement, having no particularly general solutions as far as this author is aware. Perhaps the closest is the facility in Smalltalk-80 permitting a class to reference a non-default compiler object, which takes over responsibility for interpreting the remainder of the class's definition down to Smalltalk bytecode. One limitation of this facility is that the choice of language remains with the class's author, so cannot be changed on a per-object or per-use basis, as might be desired by a particular instantiator of a class or a particular client of an object.)

5.2 Lurking metasystems

Some systems offer in-band meta-protocols for requesting particular interface versions (a common feature of Plan 9 control files). HTTP extends this to details such as the requested content language and encoding.

As mentioned previously (§3), the Unix tradition of synthetic filesystems such as /proc or Linux's /sys offer an ad-hoc grafting of specific kernel subsystems' data onto the filesystem, and in so doing, augment them with its primitive meta-level facilities (useful primarily for introspection and iteration using standard file APIs, command-line tools, shell-style scripting, etc.). A missed opportunity of some such systems is the inclusion of meta-level structure only in documentation, not programmatically. Linux's procfs manual pages provide scanf() format strings for parsing various files (such as /proc/<pid>/stat). This reflects the "large files" prejudice of Unix, and makes it impossible to write general code iterating over all attributes.

Modern /proc filesystems expose a per-process maps file detailing the memory mappings which make up a process's address space. Combined with the symbol information in loaded object files, this starts to provide a metasystem for inspecting process internals, and indeed is used to provide symbol-level backtraces. However, the most powerful such metasystem is that used for debugging. Modern Unices typically use the DWARF format [Dwarf Debugging Information Format Committee, 2010], which details compiler implementation decisions in sufficient detail to recover source-level views of programs in execution (even optimised programs, provided the compiler has generated accurate DWARF descriptions of the optimisations). It is interesting to note that this approach to debugging embodies a deeper meta-system that of Smalltalk: it documents compiler implementation decisions, down to machine level. This enables "cross-layer" debugging (switching between user-level and compiler-generated code, e.g. to track down a compiler bug). It also decouples the debugger from the debuggee, avoiding the prescriptive command language of an in-VM debug server and trivially enabling post-mortem debugging.

Extensions to the basic Unix file metamodel can be found in the use of tools such as file, which classify files based on their content, and attempts such as MIME [Borenstein and Freed 1993] at formalising such content. Such attempts so far are highly limited; in particular, the compositional nature of data encodings is not captured (as revealed by MIME types such as x-gzipped-postscript, apparently unrelated to application/gzip). Network services too are minimally and opaquely described, such as by the /etc/services, which defines a quasi-standard mapping from port numbers to protocol names (with implied semantics).

5.3 Lurkingly interposable bindings

Smaragdakis [2002] identified that ELF shared libraries embody a mixin-based composition model; mixins are a powerful primitive very similar to the "wrapper" used by Cook to model both Smalltalk- and other styles of inheritance [Cook 1989]. Its key interposition mechanism, LD_PRELOAD, is commonly used to bootstrap many other feats of interposition by overriding bindings to the C library. Applying this to the sockets API enables transparent proxying of applications, as with tsocks[9] and similar tools, while the same approach for the filesystem underlies tools such as fakeroot[10] or flcow[11] which provide clients with somewhat modified filesystem behaviour.

[9] http://tsocks.sourceforge.net

[10] http://fakeroot.alioth.debian.org/

[11] http://xmailserver.org/flcow.html

The shell makes a valiant attempt to complete unhandled portions of the Unix composition space we identified in §3. For example, bash allows commands like `diff -u <( cmd1 ) <( cmd2 )` for providing pipe-backed file descriptors where a named file is required, or /dev/tcp/<port> for redirecting to/from sockets. These approaches are limited: the latter because the shell can only introduce these "magic" filenames if the filename is interpreted by the shell (i.e. for redirection purposes), not when supplied as an argument to a program, and generally because not all functionality is invoked from a shell. Some applications reimplement shell-like facilities in their file-handling code for the same reason, but this reimplementation is both patchy and undesirable. User-level file servers such as Linux's FUSE or BSD's PUFFS [Kantee and Crooks 2007] provide a more available alternative for file redirection, effectively enabling a Plan 9-style server abstraction, albeit within a host system which does not use them so heavily to such great effect. Union mounts, a staple of Plan 9 namespace composition, are among many common use cases of these systems.

One of the most powerful late-binding devices in today's computer systems is the memory management unit. Aside from the program relocation problem it was originally designed to solve, the late binding it provides from virtual to physical addresses has enabled many other operating systems innovations (including the unified virtual memory system discussed in §3).

Bindings transmitted in message payloads are frequently rewritten with pipelined use of sed, awk [Dougherty and Robbins 1997] or Perl [Wall and Loukides 2000] as stream rewriters.

5.4 Undoing early binding

The examples we just saw all exploit inherent late-binding in the systems they compose. However, Unix applications can also bind *too early*, creating separate class of problem—"undoing" early binding. Again, many techniques for this have become mainstream. In early-bound programming languages, various dynamic update techniques have been devised [Makris 2009; Neamtiu et al. 2006]. Trap instructions and memory protection exposed by the hardware, and re-exposed in abstract form by the operating system (including BSD's mprotect()), provide useful mechanisms for intercepting early-bound code and data accesses (such as respectively for breakpoints and watchpoints). Dynamic instrumentation systems can also be used to patch bindings [Hollingsworth et al. 1997] or completely virtualise [Bruening et al. 2012] compiled code. Instrumentation techniques can also implement breakpoints [Kessler 1990] and watchpoints [Zhao et al. 2008] faster than trap-based approaches.

6. Harnessing the lurking Smalltalk

The various fragmentary techniques we have just seen suggest that building a programmable, late-bound, metasystem-enabled system can be done *using* rather than *replacing* existing Unix-based software. We use a simple running example to sketch some evolutionary additions to OS services, which generalise some subset of techniques surveyed in the previous section (although for brevity, we do not enumerate these subsets).

Suppose we wish to search some object for text matching a pattern. If the object is a directory tree of text files, our Unix command grep -r does exactly this. But suppose instead that our directory contains a mixture of gzipped and non-gzipped text files, or is actually some non-directory collection of text-searchable objects, such as a mailbox. In Smalltalk parlance, how can we make these diverse objects "understand" (via a collection of interposed objects) the messages of our grep -r (and vice-versa)?

We can view the grep process as an object which is sent a single message (execve()) with one argument: a reference (a.k.a. filename) to another object that is the root of an object graph (the directory tree). grep responds to this message by traversing this graph (again using message exchange), identifying leaf objects, obtaining a list of lines of text from each leaf, and searching the lines using a regular expression matcher. We focus on two essential operations: traversing the leaves, and reading lines of text. How can we retrofit a more object-oriented interpretation onto grep's behaviour, so that it "by default" acquires the ability to search more diverse kinds of object?

(We note that *retrofitting* is key here. Unlike more modernist approaches, we actively seek to relieve the programmer of the need to "get it right first-time". One can easily blame composition problems on developers, perhaps for misusing Unix's abstractions or failing to factor their systems appropriately. However, our postmodern viewpoint sees it as a common case for systems to start life in a specialised form, and be generalised later—an ordering which is seldom catered for in conventional programming environments.)

6.1 A metasystem spanning file and memory data

A first step in recovering an object-oriented interpretation of files and raw memory is to understand them *abstractly*, as fields instead of uninterpreted bytes. We can see this as a problem in data description. Whereas Unix files are opaque byte-streams, we wish to overlay a meta-system that can describe the higher-level semantics they encode. Existing work is relevant [Back 2002; Fisher et al. 2006], but we are more concerned with expressiveness (i.e. being able to describe both textual and binary data) than with the ability to generate correct and unambiguous parsers (indeed, some ambiguity may be inevitable). Our tentative approach is to take our lead from the metamodel embodied in DWARF debugging information, since data sent over byte-streams is invariably computed from program objects, and program objects are already described in this form. Textual encoding idioms not easily captured by DWARF, but since DWARF embeds a Turing-powerful stack machine, it is likely that few extensions are needed. In our grep example, our goal is first to capture what a "line" means (e.g. by writing DWARF describing the buffer chunk that serves as input to grep's line-by-line loop) and then to generalise out a more abstract class of "line" (something like a Collection of characters) which can be marshalled into this form for input to grep. This more abstract notion of "line", together with the marshalling step, enables diverse input data not initially structured as lines of text to be processed by grep—provided that it can be transformed to and from such an encoding. (We view the problem of selecting this transformation as a *dispatch* problem, which we visit shortly.)

6.2 A really unified binding system

grep binds to directories and files using opendir() and fopen(); our goal is to generalise these mechanisms so that they can bind to non-file objects. This is a well-trodden requirement, in that Plan 9 handles it well—assuming that the target object has been exported to the filesystem. Our goal is to minimise the effort involved in this step, so that a "found" implementation of directories—say, objects in an IMAP library instance (with IMAP folders serving as directories, and mails as files)—could be bound in to our grep's filesystem calls with little effort. We would probably not be the first to write an IMAP-to-VFS gateway; our goal instead is to investigate what meta-level information about the behaviour of an IMAP "object", say implemented in a C library, would be necessary to *synthesise* such a gateway, or at least one "good enough" for our grep purposes (i.e. to read mails line-by-line). Details such as the encoding of buffers and IMAP message formats are covered by the data description metasystem described earlier; the remaining challenge is to capture the behavioural aspects of IMAP (including IMAP protocol state and the flow of messages over the textual protocol stream) and relate them semantically to those of directories understood by grep. (We note that this kind of behavioural metadata is largely absent from a Smalltalk-style metasystem.)

6.3 Dispatch as synthesis

Returning to the simpler case of having plain grep process gzipped text, the problem appears a lot like putting a very sophisticated dynamic dispatch system into our operating system. We can say we want an fopen() call which tries to "dispatch" on the kind of data the file contains. "Dispatch" is, in fact, yet another synonym for "binding". What distinguishes this dispatch system is that unlike class- or interface-based dispatch, or even Haskell-style typeclasses [Wadler and Blott 1989], nobody should need to have told the operating system precisely how to turn gzipped text into lines. Rather, this should be derivable from a search over the meta-level information on the system's available commands, the meta-information describing the gzipped input file and the program wishing to reading it (as lines of plain text), and the set of primitive behaviours (i.e. gunzip-like functions) From this initial "fact base", we should be able to derive the need to interpose gunzip. Indeed, a first attempt at such a system might use a Prolog rule base.

6.4 Integration as an OS service

What we have just described encompasses the availability of meta-level description for both *provides* and *requires* sides of an interface, including both data and behaviour. Our hope is that working specifications can be reverse engineered out of real, existing code. For example, inference techniques for data structures [Fisher et al. 2008; Slowinska et al. 2010] could be applied to recover file formats from the programs that access them. The availability of rich metadata on both sides of an interface is game-changing, because it allows for machine-assisted or perhaps even fully-automated *synthesis* of composition-forming code. The foundations for the meta-system which enables it are already present, in vestigial form, in today's software. What this enables is integration as an operating system service. Instead of just forming bindings—which enable the flow of data—we wish to form compositions which enable the flow of *meaning*, even if the composed components do not share concrete conventions on how that meaning is encoded. Previous work has shown that bilateral relations can often be expressed much more simply in rule-based form than in general purpose programming languages [Kell 2010] and that synthesis of adequate adaptation logic appears to be a feasible approach [Yellin and Strom 1997]. Furthering these techniques, and applying them to inter-process interactions, is therefore a worthwhile target.

7. Conclusions

Smalltalk's modernist narrative holds that unification entails implementing one "unified" system—a Smalltalk system. The directions outlined in the previous section are motivated by a postmodern goal: to accept the complex reality of *existing* ("found") software, developed in ignorance of our system, and to shift our system's role to constructing *views*, including Smalltalk-like ones, of this diverse reality. This preference for building a Smalltalk *out of* the fragmented reality of today's Unix systems, rather than running isolated Smalltalks (i.e. Smalltalk VMs) each trapped *within* a Unix, parallels a contrast made by Plan 9's designers to Unix. Plan 9 was an attempt, they wrote, "to build a Unix out of little systems… not a system out of little Unixes" [Pike et al. 1990].

An implicit goal is also to unbundle this machinery from the programming language. Like any language, Smalltalk is a product of its time. Languages come and go, and appear to do so somewhat more quickly than operating systems, although we can only speculate on the reasons for this.[12] We note that meta-level facilities appear less susceptible to this "design churn" than base-level language features, because they are one step closer to a relatively small set of recurring concepts (whose recurrence though Smalltalk, Unix and Plan 9 we have been documenting). The facilites we have described in this section therefore sit most comfortably as the "waist in the hourglass", supporting diverse surface forms for languages above, and running on diverse hardware-supported "big objects" below. As such, they give a new programmer-facing role to the operating system. We have written previously about how meta-information can be used to make more interoperable language implementations [Kell and Irwin 2011], and what we have just described is effectively a generalisation of this approach. Far from being replaced by an all-conquering programming language, operating systems can and should provide the mechanisms that allow languages to come and go which maximising the composability of the software written using them. Put differently: a language is a collection of concepts that can be found and recognised within a larger system; there will be many.

Acknowledgments

I thank Michael Haupt for provoking me into writing this paper. This version has been improved by helpful comments from Michael Haupt, Peter Kessler, David Leibs, Mario Wolczko and the anonymous reviewers.

References

G. Back. Datascript – a specification and scripting language for binary data. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, pages 66–77, London, UK, 2002. Springer-Verlag.

B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN—an extensible microkernel for application-specific operating system services. *SIGOPS Oper. Syst. Rev.*, 29(1):74–77, Jan. 1995.

N. Borenstein and N. Freed. RFC 1521: Mime (multipurpose internet mail extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. IETF Request for Comments, 1993.

D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1176-2.

W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, Providence, RI, USA, 1989.

D. Dougherty and A. Robbins. *Sed and Awk*. O'Reilly Media, Inc., 1997.

D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, page 78, Washington, DC, USA, 1995. IEEE Computer Society.

K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 2–15, New York, NY, USA, 2006. ACM.

K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 421–434. ACM, 2008.

Dwarf Debugging Information Format Committee, 2010. *DWARF Debugging Information Format version 4*. Free Standards Group, June 2010.

[12] One slightly tongue-in-cheek explanation might be that rewriting device drivers is more onerous than rewriting user-level software. David Leibs noted (in personal communication) that in the years following Ingalls' Byte Magazine article, the Smalltalk project began to embrace Unix, precisely to improve the range of hardware on which Smalltalk could run.

R. A. Gingell, J. P. Moran, and W. A. Shannon. Virtual memory architecture in SunOS. In *Proceedings of the USENIX Summer Conference*, pages 81–94. USENIX Association, 1987.

A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

J. Hollingsworth, O. Niam, B. Miller, Z. Xu, M. Goncalves, and L. Zheng. Mdl: a language and compiler for dynamic program instrumentation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 201–212. IEEE, Nov 1997.

G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, Apr. 2007.

D. H. Ingalls. Design principles behind Smalltalk. *Byte Magazine*, 6(8): 286–298, 1981.

A. Kantee and A. Crooks. Refuse: Userspace fuse reimplementation using puffs. In *Proc. of the 6th European BSD Conference (EuroBSDCon)*, 2007.

S. Kell. Component adaptation and assembly using interface relations. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 322–340, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6.

S. Kell and C. Irwin. Virtual machines should be invisible. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, &#38; VMIL'11*, SPLASH '11 Workshops, pages 289–296, New York, NY, USA, 2011. ACM.

P. B. Kessler. Fast breakpoints: design and implementation. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming Language Design and Implementation*, PLDI '90, pages 78–84, New York, NY, USA, 1990. ACM. ISBN 0-89791-364-7.

T. J. Killian. Processes as files. In *USENIX Summer Conference Proceedings*. USENIX Association, 1984.

S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the USENIX Summer Conference*, volume 86, pages 238–247. USENIX Association, 1986.

I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *Selected Areas in Communications, IEEE Journal on*, 14:1280–1297, 1996.

K. Makris. *Whole-Program Dynamic Software Updating*. PhD thesis, Arizona State University, December 2009.

P. Mochel. The sysfs filesystem. In *Proceedings of the Linux Symposium, Volume One*. Linux Symposium, 2005.

I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on programming language design and implementation*. ACM, 2006.

J. Noble and R. Biddle. Notes on postmodern programming. Technical Report CS-TR-02-9, Victoria University of Wellington, Wellington, New Zealand, 2002.

R. Pike, D. Presotto, K. Thompson, H. Trickey, et al. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*, 1990.

R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Orr, and R. Sanzi. Mach: a foundation for open systems [operating systems]. In *Workstation Operating Systems, 1989., Proceedings of the Second Workshop on*, pages 109–113, 1989.

D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Commun. ACM*, 17:365–375, July 1974.

T. Roscoe. CLANGER: an interpreted systems programming language. *SIGOPS Oper. Syst. Rev.*, 29(2):13–20, 1995. ISSN 0163-5980.

M. Schelvis and E. Bledoeg. The implementation of a distributed Smalltalk. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*, volume 322 of *ECOOP '88*, pages 212–232. Springer Berlin Heidelberg, 1988.

A. Slowinska, T. Stancescu, and H. Bos. DDE: dynamic data structure excavation. In *Proceedings of the first ACM Asia-Pacific workshop on systems*, pages 13–18. ACM, 2010.

Y. Smaragdakis. Layered development with (Unix) dynamic libraries. In C. Gacek, editor, *Software Reuse: Methods, Techniques, and Tools*, volume 2319 of *Lecture Notes in Computer Science*, pages 33–45. Springer Berlin Heidelberg, 2002.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

L. Wall and M. Loukides. *Programming Perl*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2000. ISBN 0596000278.

D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19:292–333, 1997.

Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: efficient debugging using dynamic instrumentation. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, CC'08/ETAPS'08, pages 147–162, Berlin, Heidelberg, 2008. Springer-Verlag.

H. Zimmermann. OSI reference model: The ISO model of architecture for open systems interconnection. In C. Partridge, editor, *Innovations in Internetworking*, pages 2–9. Artech House, Inc., Norwood, MA, USA, 1988.