Critique of 'Files as Directories: Some Thoughts on Accessing Structured Data within Files' (2)

Stephen Kell
Department of Computer Science and Technology
University of Cambridge
Cambridge, United Kingdom
stephen.kell@cl.cam.ac.uk

ABSTRACT

In this critique I argue that the motivations and direction of the 'files as directories' idea are sound, but the conceptual difficulties are considerable yet non-obvious, and are not limited to those identified by the author. I highlight a selection of concerns, including Unix's latent pluralism, the blurred boundary between naming and computation in languages, and issues of bidirectionality, semantic diversity and support for economical migration.

CCS CONCEPTS

• **Software and its engineering** → **File systems management**; *Abstraction, modeling and modularity*; • **Human-centered computing** → **Command line interfaces**;

KEYWORDS

Unix, pluralism, Plan 9, Smalltalk, files, objects

ACM Reference Format:
Stephen Kell. 2018. Critique of 'Files as Directories: Some Thoughts on Accessing Structured Data within Files' (2). In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion)*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3191697.3214325

1 INTRODUCTION

Raphaël Wimmer's 'Files as Directories' [Wimmer 2018] is an exploratory piece which considers the possibility of further generalising the Unix-like filesystem abstraction such that it exposes structure *within* what would presently be individual files, serving this structure in the form of a directory tree. Accesses to this tree, using the conventional filesystem interfaces and/or tools targeting them, could achieve what currently requires use of format-specific library APIs. In this way, structured data could be made more accessible, especially to end users or inexperienced programmers. Such access to data might also prove more straightforward even for expert users.

<Programming'18> Companion, April 9–12, 2018, Nice, France

I found myself in broad agreement with the aims of this work. This critique analyses and complements the arguments, picking out somewhat differing emphases and proposing some slightly different conclusions. In short, it contends that the true difficulties with realising this concept are relatively non-obvious and have been downplayed in the author's presentation. I offer my perspective not to criticise the idea, which I hold to be a good one, but to help focus attention on future efforts most likely to make it a success. I begin with a brief recap of the strengths and weaknesses of the concept, contrasting the author's perspectives with my own.

2 CONCEPTUAL STRENGTHS

As evident in the earliest presentations [Ritchie and Thompson 1974], the filesystem in a Unix-like operating system is a space of interaction between the user and the system. The most basic Unix skills involve use of the shell to create and control programs, including the use of named files. The filesystem is also a space of interaction between distinct programs, even outwith the user's direct control, reading and writing files that are (usually) looked up by name. By bridging the programmatic and the interactive, the filesystem occupies a powerful position. The more objects are exposed through the the filesystem, the more both programmers and interactive users are empowered. (These populations are, of course, far from disjoint.)

It is therefore fitting that the author arrives at the issue of filesystems from the viewpoint of interaction in general, and end-user programming in particular. The transition from interactive use, via customisation using textual notations, to fully fledged programmer, is well observed. So too are the author's examples of structured data which could usefully be exposed as directory structures within the 'files as directories' (FAD) concept: the copy-paste clipboard; comma-separated (CSV) records; section-structured textual configuration files; e-mail messages; and JPEG images. Using these, he identifies problems with status-quo practices around client library APIs (e.g. libjpeg, for the JPEG example): customisation barriers faced by end users; limited API availability when working with a given kind of structured data, only through language-specific APIs; limited learnability of APIs by end users; limited forwards compatibility, cf. the desire for old applications to support new file types without modification; and limited interactive explorability of structured data. All of these would be ameliorated if the relevant structure were served as a filesystem rather than via APIs.

Some of the other motivations listed are less convincing. The suggestions that access control and network transparency are acquired 'for free', or that concurrency is aided by this arrangement, raise more questions than they answer. If a file format does not provide for access control, the metadata will require separate storage; alternatively, if it does provide notions of access control, a mapping is required between the format's model and the system's, with ensuing complexity. Network transparency, while desirable, is hard to achieve: effective network-transparent abstractions require a carefully crafted partitioning of operations and state across different locations in the network, to mitigate high variances in available latency or throughput (echoing the latency concerns in the author's own quotation of Theodore Ts'o). Regarding concurrency, even if disjoint parts of a file are being accessed, conflicts are likely, since binary file formats often include whole-file layout or summary information such as a 'length in bytes' field describing a variable-length sequential structure. Even under updates to disjoint elements of that sequence, a conflict arises on the necessary update to that summary information. These problems, while far from insurmountable, must be recognised as such.

3 CONCEPTUAL WEAKNESSES

In the foregoing dissection of the more doubtful strengths of the proposal, a more general phenomenon is visible: that a full implementation of the filesystem abstraction represents a much more challenging programming assignment than an ad-hoc API that is free to impose simplifying constraints. Obviously there are also benefits to be reaped from such an investment of implementation effort, in the form of a much more flexible and uniform user-facing interface, but the reward curve may or may not be favourable. Since, as it stands, the proposal implies an invitation to rewrite client libraries as filesystems, these economics matter (see §8).

This problem of mappings is recurrent: how to define a good enough mapping, and transfer sufficient meaning through it—including the avoidance of spurious transfers. Indeed, the author notes this difficulty with respect to 'sensible semantics', 'conversion losses', error messages and ordered structures.

The author also cites composability, with reference to the pipe-and-filter style. If more data were exposed in the filesystem, would this engender a more compositional style among clients? This is questionable, since pipes and filters in Unix work over byte streams, not directory trees. Indeed, the concept of 'filesystem' in Unix often packs together these two rather different abstractions: a namespace-based file access API dealing in hierarchical structures (files denoted by path). and a descriptor-based read/write API dealing in distinctly flat structures (files denoted by descriptor). There is an air of equivocation to the argument here: if anything, compositional programming, at least in the pipe-and-filter mode, appears in fact rather easier with byte-streams (files) than with (files as) directories, since one can pipe a byte-stream to a program which interprets it. The nearest analogue with directories is to create a succession of temporary trees which may be passed by name to a client program. The latter moves away from the expression-oriented effect-free simplicity of pipelines, towards a style that brings the resource management complexity of naming, creating and deleting the intermediate trees. (To avoid these, one might quickly be tempted to encode such a tree in a byte-stream format such as `tar`!) In the pipe-and-filter case it seems that compositionality emerges not from directory structure, but the insertion of expression-oriented computation (piping to a program amounts to a function application) interchangeably with storage (redirecting to a file). The desired affordance appears to be not directories per se, but the ability to swap straightforwardly between a byte-stream and a directory 'view' of the data, depending on what suits each stage of processing.

I'll revisit three of the preceding conceptual challenges in subsequent sections: the desirability of some pluralist notion of 'views' that permit selection of appropriate abstractions in appropriate contexts (§4, §6). the challenge of defining mappings with semantic consistency (§7), and the necessity of a favourable investment/reward curve in (more complex) reengineering (§8). In so doing, we will also uncover a question (§5) about what a 'naming language', such as the filesystem's language of pathnames, really is.

4 PLURALISM

The author anticipates several difficulties with realising FAD: low-level compatibility issues, performance issues, semantic limitations, poor error reporting, conversion losses, and loss of ordering semantics. In the sections that follow, I will mostly focus on an alternative selection of limitations, some non-obvious, that I believe at least as important. I mention these not because they render the idea unworkable—I don't believe they do—but to set out the design challenges they imply.

The first challenge is pluralism—a seldom-remarked yet critically important property of Unix, as I have argued elsewhere [Kell 2013]. Unix is pluralist partly by accident, thanks to its minimalism. For example, by leaving the command language out of the kernel, it permits many different shells with their own built-in commands, and allows user to extend the command language simply by installing additional binaries. Similarly, by omitting certain abstractions—such as record-structured files, for example—it becomes naturally amenable to multiple ways of breaking down the same data, where no single way has dominance or favour. This is often realised with pipelines, in which multiple programs may be interpreting and reinterpreting the same data in different ways. For example, even a very pedestrian pipeline might combine character-wise (e.g. `tr`), line-wise (e.g. `sed`) and whole-file (e.g. `sort`) operations in one go. Intervening commands (often, again, `tr` or `sed`) might logically restructure the file again and again by rewriting its delimiters or separators. And of course, the support for many source languages arises from avoiding any deep special favours to the implementers' own C language.

It seems reasonable that proposals seeking to extend Unix should preserve these kinds of pluralism. Here we can consider

what this might mean for a file containing structured data. At its simplest, a file named

/path/to/ file

... might be viewed as a directory simply by the designation

/path/to/ file /

... but then, by implication, there is a *single* interpretation of that file as a directory. This rather follows GNU Hurd's 'translator' concept[1], in which a filesystem node has a stored 'translator' which may be set by the user, and otherwise persists, but only has one value at any time. Giving favour to a single interpretation, fixed by the administrator or file owner, appears restrictive and is definitely not pluralist.[2]

Alternatively, one can take the approach of avfs[3], which uses filename suffixes like the following.

/path/to/ file #utar

Here 'utar' refers to a function which interprets the file as a tape archive (tar) file. In other words, the choice of interpretation has been left to the client, i.e. the user/programmer naming the file. That choice is not free, however; a fixed repertoire of such functions is defined by the AVFS system. Arguably a conceptually cleaner design, and certainly one that is both pluralist and the most 'open', would also permit something like

/path/to/ file #/path/to/my/utar

... where a user-supplied interpretation is applied by addressing it within the filesystem, potentially itself by a FAD-denoting expression. In other words, file-as-directory interpretations become just another kind of object that may live in the filesystem, much as shell commands did in the original Unix design. Unlike command invocations, the result of a FAD's invocation is itself necessarily a named filesystem entity.

5 NAMING LANGUAGES

What we have just seen is a transition from a language whose phrases (pathnames) have a linear structure (/path/to/file) into one with a branching structure (/path/to/file#/path/to/my/utar). The latter admits expressions of the form 'apply this to that', which the former does not. This tree structure appears to be a demarcation of traditionally non-computational 'naming' languages (pathnames, selectors, coordinates) from more general computational 'denoting' languages (including human-facing programming language, but also lambda calculus, SKI combinators, or any Turing-powerful language). So-called 'elegant' designs seem to select only *one* such application primitive each, whether function call, beta reduction, or message send. The trajectory of other designs has also been to migrate towards ever fewer 'verbs': Plan 9 narrowed the interface of Unix in favour of deeper namespaces (e.g. control files instead of ioctl()). The REST conception of HTTP [Fielding 2000] advocated a small collection of verbs (get, put, post, delete) while pushing semantic structure into the object space itself (making more complex operations be nameable 'resources' in their own right).

Another demarcation of naming languages from programming languages might be the resource expectations: naming languages in a storage-only filesystem necessarily denote objects that already exist in manifest form, so can be accessed in time linear (roughly speaking) in the length of the name. By contrast, given a phrase denoting a computed value, in general we expect that value need not exist, and the process of 'resolving' the name involves computing its manifestation—for which the resource demands are highly variable.

It is unclear whether this distinction, between naming languages and (computational) programming languages, withstands an overriding concern for usability. A system in which pathnames *can be* complex computational expressions is likely rapidly to become in in which they actually frequently *are*, perhaps in ways deleterious to comprehensibility and usability. This contrasts with the linear simplicity of pathnames; since much of the human population struggles to grasp even simple hierarchical abstraction, this problem may be deeper than most computer scientists realise. Nevertheless, the potential orthogonality and expressiveness of a rich naming language brings benefits as well as drawbacks. Are pluralism and 'free interpretation' (to paraphrase FAD itself), taken together, too much to pack into an intuitive, novice-friendly system? This is a design question meriting considerable research on its own.

6 BIDIRECTIONALITY

A further semantic issue is that of two-way data flow: reading and writing through the same interface. Storage interfaces permit this by definition. Can arbitrary files do so? For example, can we create new files in our utar'd directory, hence adding content to the archive? In general, such questions imports the 'view update' problem of database lore [Furtado and Casanova 1985], also studied in programming languages research such as in the 'lenses' of Foster et al. [2005]. Concurrent updates complicate matters still further.

7 THE INEVITABILITY OF METASYSTEMS

As the author notes, this trajectory of generalising the 'file' abstraction has been pursued before, notably in the Plan 9 operating system [Pike et al. 1991]. This refined and simplified the Unix filesystem interface considerably, extended it to new depths of generality (embracing network protocols, windowing system entities and more) and, overall, adopted the serving and consumption of named files as the principal interprocess communication model. File servers, being ordinary user programs, had something in common with our suggestion earlier that utar should simply be a program supplied

[1] This is described at https://www.gnu.org/software/hurd/hurd/translator.html as retrieved on 2018/5/10.
[2] Using out-of-band stored attributes to record such translator assignments appears also a failure to unify around the filesystem; why are these attributes not themselves simply files?
[3] A Virtual File System, a Linux-based open-source project active since at least 2002 until the time of writing, at http://avfs.sourceforge.net.

(or at least named) by the user. Operations not naturally expressible on storage-style files, such as ejecting a disc or launching a missile, would in Plan 9 be represented as 'control files' with no payload state but on which reading or writing some data, often as a request-response pair, would trigger some arbitrary procedural action. Although powerful and pragmatically convenient, such moves come at the expense of semantic uniformity. What does it mean to copy a filesystem? With plain old files it is obvious, but with such behaviourally diverse entities, it is no longer clear—nor is taking a copy any longer an obviously 'safe' operation (especially where missile launchers are concerned). Even when one is limited simply to structured data, not control files, non-local effects such as integrity constraints ('if X exists, Y must also exist') can stymie a generic copying algorithm that works fine on files.

Modelling semantic diversity, and also commonality, is the role of metasystems. Consider Smalltalk's system of classes. A class conveys how two objects behave alike, and also how they may differ. A hierarchy of classes conveys (ideally) refinement relations on these behavioural contracts. For example, copying is only permitted for objects whose class defines a method for that purpose. Such information clearly assists in writing a robust generic copy. In a system of rich structured data, it seems inevitable that some sort of meta-level interrogation should be necessary. Of course, the strength of this approach is that both views are available. Like in dynamic languages, in a filesystem world it is the 'default' base-level view that is the generic one, exposing similarity: ordinary programming is simply sending messages to objects, or invoking operations on files, all of which are *a priori* equivalent. It is up to the meta-level, as a 'splitter', to supply the distinctions, by capturing distinct classes of object. By contrast, statically typed languages are 'splitters' at the base level, with each syntactic phrase constructed inhabiting a potentially distinct classification (static type). The role of a meta-level interface is generally to recover commonality—for example, iterating uniformly over all functions in a module, in spite of their differing types, so acting as a 'lumper' (e.g. to enable generative metaprogramming).

In the filesystem case, the basic abstraction is a 'lumper', cutting universally across all objects, and it is only the meta-level that splits by specialism. Although this is not so far from a Smalltalk-like model, the notion of messages 'understood' remains difficult: does a control file really 'understand' a `write` message, if its behaviour is nothing to do with writing? Whereas in Smalltalk this would happen only in rare cases of coincidence (choosing the same message name for semantically different operations), in a shoehorned filesystem, whether Plan 9 or (to some extent) FAD, this happens intentionally: 'write' is consciously overloaded with some sort of distinct meaning. The potential for obscure, confusing interfaces is therefore greater, and will require mitigations—perhaps in FAD implementations (ensuring only write-like operations are exposed as `write()`), or perhaps, more polymorphically, in clients: ensuring that a program such as `cp`, that is trying to read or write files, is doing so with a uniform notion of what those operations will do, and will abort on divergence (or perhaps simply ignore divergent files). Such tests for 'divergence' or 'write-likeness', if they are tractable at all, will require considerable care at both specification and implementation.

8 REPURPOSING EXISTING CODE

Given a world with FAD support, a naïve next step is simply to 'rewrite all the libraries'. Is it possible instead to recover filesystem interpretations of files without rewriting huge volumes of code? This is doubly important since, as I noted earlier, a filesystem-like view of data is likely to present a more complex, more general interface than some ad-hoc domain-specific API, so a favourable investment/reward curve will be necessary.

The ideal curve would offer an automated or semi-automated path to migrate from client library to filesystem. Although this is an ambitious goal, there is reason to be cheerful. Recovering structure implicitly is a theme of much under-represented research in disparate venues; a now-dated selection springs to mind [Cozzie et al. 2008; Fisher et al. 2008; Slowinska et al. 2010] and in the age of rapidly advancing machine learning, there is considerable opportunity for new techniques. Naturally, simply learning the desired logic in its entirety would be the ideal, but more prosaically, learning 'metaformat'-based descriptions, concisely and declaratively capturing the structure of binary data may be a useful stepping stone. The metaformats are exemplified by Infra of Hall et al. [2017] in the case of stored data, and by various notions of debugging-oriented 'type' metadata in the in-memory case. Unlike its namesake type information in a language runtime, this kind of information describes not only types' identities or definitions, but also their encodings and layouts, i.e. implementation-level details about in-memory representation [Free Standards Group 2010; Kell 2015].

Unifying file-based and memory-based metaformats is a potential technique here. (Indeed, the design of Infra anticipates this idea.) Since a client library is necessarily software which decodes a structured file into more primitive manifest elements *in memory*, such debugging-derived meta-level descriptions of its input and output memory buffers may already represent a job part done. There is then an intriguing possibility of semi-automated, perhaps trace-guided refactoring-style tools to infer and extract metaformat-based descriptions from the pre-existing client libraries that embody these formats implicitly. The desired curve might be achieved by gradually morphing such code, likely with human assistance, into a workable filesystem implementation. Recent work on automated transplantation [Barr et al. 2015] or older work on less automated rule-based approaches [Bracciali et al. 2005; Kell 2010; Purtilo and Atlee 1991] may transfer partially to this task. The decomposition of memory into a hierarchy of *allocations*, maintained in the `liballocs` runtime [Kell 2015], and encompassing both program state and memory-mapped files, may also provide a useful intermediary for relating in-memory with on-disk byte patterns.

9 CONCLUSIONS

Files as directories is a powerful idea with a long history and a potentially impactful future. The most significant technical challenges to realising its goals are not necessarily the most obvious. In this critique I have drawn attention to some less obvious concerns: Unix's latent pluralism, naming as a computational language, bidirectionality, metasystems, and economic incentives.

That is not to detract from the considerable merit of the idea and the importance of follow-up in this direction. The most significant *non*-technical challenges are in acceptance of such a boundary-spanning concept by one or more established research programmes. It is appropriate that this research is motivated in the context of interaction, and initially hammered out by argument and criticism (as the present venue uniquely allows). Despite obviously being a wide-reaching matter of system design, such issues span beyond what is typically considered under the present-day research heading of 'systems'. Similarly, despite also being a matter of programming, it is liable to be perceived as similarly off-topic in many venues matching that keyword. Conceptual exploration, argument and criticism go hand-in-hand with system-building; I look forward to the future iterations, both practical and conceptual, that this work might yield.

ACKNOWLEDGMENTS

I am grateful to Raphaël Wimmer, for his thought-provoking article, and the participants of the Salon des Refusés workshop 2018 for the discussions that have further shaped and refined this critique.

REFERENCES

Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 257–269. https://doi.org/10.1145/2771783.2771796

A Bracciali, A Brogi, and C Canal. 2005. A formal approach to component adaptation. *J. Syst. Softw.* 74 (2005), 45–54.

Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. 2008. Digging for data structures. In *In Proceeding of 8th Symposium on Operating System Design and Implementation (OSDIâĂŹ08*.

Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures*. Ph.D. Dissertation. University of California, Irvine. http://portal.acm.org/citation.cfm?id=932295

Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. 2008. 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)*. ACM, New York, NY, USA, 421–434. https://doi.org/10.1145/1328438.1328488

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2005. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on principles of programming languages*. ACM, New York, NY, USA, 233–246. https://doi.org/10.1145/1040305.1040325

Free Standards Group. 2010. *DWARF Debugging Information Format version 4*. Free Standards Group.

Antonio L. Furtado and Marco A. Casanova. 1985. Updating Relational Views. In *Query Processing in Database Systems*. Springer, 127–142.

Christopher Hall, Trevor Standley, and Tobias Hollerer. 2017. Infra: Structure All the Way Down: Structured Data As a Visual Programming Language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, New York, NY, USA, 180–197. https://doi.org/10.1145/3133850.3133852

Stephen Kell. 2010. Component adaptation and assembly using interface relations. In *Proceedings of 25th ACM International Conference on Systems, Programming Languages, Applications: Software for Humanity (OOPSLA '10)*. ACM.

Stephen Kell. 2013. The Operating System: Should There Be One?. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems (PLOS '13)*, Tim Harris and Anil Madhavapeddy (Eds.). ACM, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/2525528.2525534

Stephen Kell. 2015. Towards a Dynamic Object Model Within Unix Processes. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 224–239. https://doi.org/10.1145/2814228.2814238

Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, Tom Duff, and Gerard Holzmann. 1991. . Technical Report CSTR 158. Bell Labs.

JM Purtilo and JM Atlee. 1991. Module Reuse by Interface Adaptation. *Software - Practice and Experience* 21 (1991), 539–556.

Dennis M. Ritchie and Ken Thompson. 1974. The UNIX time-sharing system. *Commun. ACM* 17 (July 1974), 365–375. Issue 7. https://doi.org/10.1145/361011.361061

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

Raphaël Wimmer. 2018. Files as directories: Some thoughts on accessing structured data within files. In *Companion to the Second International Conference on the Art, Science and Engineering of Programming (Programming '18)*. ACM, New York, NY, USA.