Making system configuration more declarative

John DeTreville
Microsoft Research
johndetr@microsoft.com

Abstract

System administration can be difficult and painstaking work, yet individual users must typically administer their own personal systems. These personal systems are therefore likely to be misconfigured, undependable, brittle, and insecure, which restricts their wider adoption. Because updating the configuration of today's systems involve imperative updates in place, a system's correctness ultimately depends on the correctness of every install and uninstall it has ever performed; because these updates are local in scope, there is no easy way to specify or check desired properties for the whole system. We present a more checkable declarative approach to system configuration that should improve system integrity and make systems more dependable. As in the earlier Vesta system, we define a *system model* as a function that we can apply to a collection of system parameters to produce a statically typed, fully configured *system instance*; models can reference and thereby incorporate *submodels,* including submodels exported by each program in the system. We further check each system instance against established *system policies* that can express a variety of additional *ad hoc rules* defining which system instances are acceptable. Some system policies are expressible using additional type rules, while others must operate outside the type system. A preliminary design and implementation of this approach are under way for the Singularity OS, and we hope to specify and check a number of ad hoc system properties for Singularity-based personal systems.

1. Terminology and introduction

Programmers write programs; *administrators* configure these programs into systems; *users* apply these systems to their tasks.

Some individuals combine these roles, but most do not. For instance, some expert users are also expert programmers but most are not; most users rely instead on the large available body of general-purpose programs written by others.

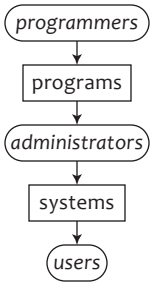

Figure 1. Programmers, administrators, and users.

Similarly, some expert users are also expert administrators but most are not. System administration can be difficult and painstaking work; programs in a system can interact in unexpected ways, and installing one program can very readily break another. Unfortunately, the users of personal systems must typically administer their own systems, and we believe that this creates barriers to the wider adoption of new personal systems.

As a result, a great many personal systems are poorly configured and poorly maintained. They do not work well. They are undependable. They are brittle. They are insecure. How might we do better?

2. What can go wrong?

Let's consider a (very) simple example. The user, acting as the *de facto* administrator, chooses four programs—photo editor E, camera driver C, printer driver P, and kernel K—and configures them into the system shown in Figure 2. What can go wrong?

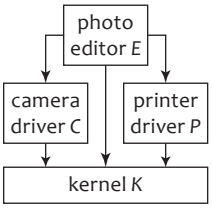

Figure 2. One system configuration.

- The user can choose programs that simply do not work together—at all. Printer driver P may require a formatting language that photo editor E cannot produce, or produces incorrectly. If there are multiple versions of P and E, the user can choose a bad pair.
- The user can misconfigure the programs, causing them not to work together. Misconfiguring UTF-8 support in kernel K, let's say, might change its semantics enough to break its clients.

Most existing configuration tools are imperative in nature. The system configuration exists as mutable state in the file system, in the Windows registry, *etc.,* and the *de jure* or *de facto* administrator updates the configuration in place by installing and uninstalling programs. A system's correctness therefore depends on the correctness

and the appropriateness of each install and uninstall the system has ever performed, as well as their exact order.

- As the system changes over time, even an initially good system can become misconfigured. Since many configuration settings are shared or global, local updates to one component or setting can readily create problems elsewhere.

The result is that configuration management in current systems falls short in several ways.

- System configurations are *brittle, imperative,* and *history-dependent,* especially since our tools for managing configurations can deal only with *local* constraints. Let's imagine that installing program C or P must reconfigure K. If we install C, then P, is K still correctly configured for C? If we uninstall C, is K still correctly configured for P?
- System configurations are *imprecise* and overly *dynamic.* When a program uses another program—perhaps as a library, perhaps as a service, perhaps otherwise—the system can choose the other program in some arbitrary fashion at runtime, such that we cannot check the combination statically.
- System configurations are *insecure.* When a system boots, it runs whatever system it finds on its hard drive, with no opportunity for enforcing an end-to-end check.

3. What has been tried?

There are several existing approaches for improving the task of system configuration, each with its own shortcomings.

3.1. Central administration

Central administration works well in many enterprise environments, where expert professional administrators can create some small number of standard configurations. These central administrators can choose, customize, and configure programs to work well together, and maintain the resulting configurations over time.

Central administration seems much less suitable for personal systems. Home systems, for example, are quite varied and quite frequently reconfigured. As other personal systems, such as mobile phones, become more like home systems, they also become less amenable to central administration. We therefore should not expect central administration to work well for personal systems.

We also propose that, all else being equal, users' interests are best served when they can choose their own programs [17].

3.2. Closed systems

A similar approach is the *closed system,* where a system's programs all come from a single supplier or integrator. Closed systems are common in the world of consumer electronics, where the manufacturer delivers and upgrades a typical system's firmware monolithically.

Most existing personal systems are not closed, except for the very simplest, and closed systems seem less and less suited over time to satisfy the ever-expanding needs and expectations of individual users. Simple closed systems cannot necessarily scale to serve complex, varied environments.

3.3. Stronger isolation

Can we factor our open systems into some number of closed programs that do not interact? Each program might execute in a separate virtual machine or virtual environment without interfering with the others.

No. Real programs interoperate with each other. Program C copies photos from a digital camera; E edits them; P prints them. Reducing extraneous interaction between programs can reduce interference, of course, but real programs will always interact. We must allow users to choose their programs independently, even though these programs can and will interact.

3.4. Stronger interfaces

Many systems let programs interact only across strongly typed interfaces. Strong static typing can eliminate many mismatches and misconfigurations, but it is not a panacea; a program A can work perfectly well with B but not at all with the identically typed B′, and then again with B″.

Some bad configurations won't type-check, but many more will have subtler problems. We need solutions that are more powerful than strongly typed interfaces as they currently exist.

3.5. Better programs

If program P works with K but not with K′, doesn't that mean that K satisfies its contract and K′ does not?—or that P is somehow depending on unspecified behavior? Can't we just write P, K, and K′ correctly in the first place?

No, in general, we can't. We believe that our programs will continue to have bugs, and our interfaces will continue to elide important information. We will continue to integrate programs from different programmers with different assumptions, and we will continue to discover their interfaces and requirements experimentally. In short, we will continue to integrate imperfect programs for the near future, and perhaps longer.

3.6. Smarter installers

Some installers can explicitly model programs' dependence on each other, eliminating some misconfigurations; examples include Windows Installer [12] and the

Debian package management system [1]. If P and K′ do not work together, installing P might also upgrade K′ to K″, while upgrading K to K′ should perhaps fail if P is already installed.

Existing installers of this sort typically can check system configurations for local consistency but not for global consistency. They can avoid some misconfigurations but not others.

3.7. Smarter users

Many people argue that users should better understand the internal workings of their systems, and that administering their systems helps them learn. If users learn enough about how their systems work, the argument goes, then they can configure their systems as they see fit. Conversely, if users don't really understand their systems, then they get what they deserve.

We argue the opposite. Eliminating the need for users to administer their own systems should be as beneficial as eliminating the need for them to develop their own programs. Most users—who are neither expert programmers nor expert administrators—are better off when others can perform these tasks for them.

3.8. Forensic tools

Since occasional system misconfigurations seem inevitable, it is useful to provide tools for diagnosing and undoing misconfigurations when they occur [18] [15]. Even so, it would seem preferable to catch misconfigurations at earlier stages, before falling back on forensic tools.

4. Declarative configuration

We propose a *declarative* approach to system configuration that addresses many of these problems. Our proposal derives from the earlier Vesta software configuration system [5] [6], which itself derived from the Cedar System Modeller [10].

- We compose a declarative *system model* that completely and precisely specifies the system as a whole.
- Evaluating the system model, as applied to the *system parameters,* produces a complete, fully configured *system instance*.
- Extending the Vesta approach, we can further check each system instance against established *system policies* that can express a variety of ad hoc rules that define which system instances are acceptable.

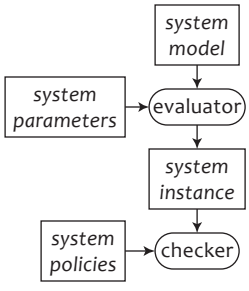

Figure 3. Models, parameters, instances, and policies.

We argue that this declarative approach to system configuration can improve the integrity and thus the dependability of personal systems. (Other analyses of the problems of system administration have also focused on mutable configuration state [7]; our declarative approach can eliminate much of this mutable state.) A preliminary design and implementation of this approach are under way for Singularity, a new research OS intended to support the construction of dependable systems [8] [9].

4.1. Models

Models are hierarchical. The system model can reference—and thus incorporate—any number of submodels, usually including one for each component program, and these submodels can themselves be hierarchical. Programmers, publishers, and remote administrators can write these submodels, while the local administrator composes them into the local system model. Our goal when writing system models and submodels is to express rules for how we can correctly compose the various programs into systems. Our hope is that system models can be easy to compose from their submodels.

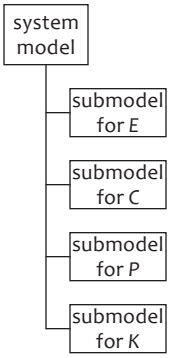

Figure 4. A system model and its submodels.

In our example, the system model incorporates submodels for programs E, C, P, and K. We apply each program model to its appropriate parameters to yield a program instance, and we compose these program instances into a fully configured system instance.

Let's consider kernel K from Figure 2. (We present these examples in the functional language Haskell [13] [4], although our implementation for Singularity may not itself use Haskell.) A program instance exports some number of values. The kernel instance in our example (examples are partially elided in this paper) exports the kernel's identity (a secure hash of type `Hash`) and a *reboot* operation (of type `KReboot`).

```
> data K = K Hash KReboot
```

The function `kModel` is our kernel model. It takes no parameters, and returns a kernel instance of type `K`.

```
> kModel ()
>    = K (Sha256 "b6f8…2ab7") doKReboot
```

(This partially elided hash identifies the binary for kernel K. A more realistic example might return different hashes depending on its parameters.) Here, `doKReboot` implements the *reboot* operation for kernel K.

We define the types `C`, `P`, and `E`, and the functions `cModel`, `pModel`, and `eModel`, similarly.

Finally, the data type `System` represents the system instance; it exports its secure hash (of type `Hash`) and a

run operation (of type SRun), along with its component program instances.

```
> data System = System Hash SRun K C P E
```

The system model systemModel is a function that takes the four program instances (of types K, C, P, and E) and returns a system instance (of type System).

```
> systemModel k c p e
>   = System
>       (bind [hash k, hash c,
>              hash p, hash e])
>       doSRun k c p e
```

The function bind links a number of programs, identified in this example by their secure hashes, and returns the secure hash of the result; doSRun is a function that implements the system's *run* operation.

4.2. Evaluation

Applying a model to its parameters evaluates to an instance. We produce instances k, c, p, e, and system of types K, C, P, E, and System.

```
> k = kModel ()
> c = cModel k
> p = pModel True k
> e = eModel True k c p
> system = systemModel k c p e
```

(Here, pModel and eModel each take one extra Bool parameter.) The resulting value system is the fully configured system instance, which exports k, c, p, and e.

Model evaluation has no side effects, and applying the same system model to the same parameters always produces the same system instance. We can produce a new system instance from an updated model, or from an old model with updated parameters, but we always produce it functionally, and not as a local update to the current system instance on the current machine.

The functional nature of model evaluation is convenient for system administrators, especially for the administrators of distributed systems. For example, it lets us produce system instances on systems other than the ones on which they will run. It might be much simpler to construct a new system model for a light switch on a personal computer or some similarly powerful system than on the light switch itself.

4.3. Type checking and subtyping

Not only are our system instances and program instances values, they are also statically typed and statically checkable. Our models, *etc.,* are also statically checkable.

In our example, a system instance of type System must contain a kernel instance of type K, and a system instance will not type-check if another type is used. When this is too constraining—perhaps we would like to use a kernel of type K′ that also exports a *shutdown* operation, so that K′ <: K—we can use *subtyping* to express looser rules. Here, we redefine System to include any kernel type k that exports at least a *reboot* operation, as defined by the HasKReboot type class. (Belonging to a Haskell type class is like implementing a C# or Java interface.)

```
> data System
>   = forall k .
>       HasKReboot k
>       => System Hash SRun k C P E
```

(Here, k is an existentially quantified type variable.) We also declare our own type K to belong to the type class HasKReboot. We can make similar changes elsewhere in our example to take further advantage of subtyping.

4.4. Installation

Installing a new system instance involves three steps.

1) We make the new system instance available on the local machine or across the network.
2) We make the new system instance *current* by setting the local machine to boot only from that instance, as specified by the instance's secure hash.
3) We atomically reboot the local machine.

(We expect that we can eliminate the reboot in many cases.) More than one system instance can be available at once—and they can share common structure—but only one can be current at a time.

We provide no way for an installer or an administrator to modify a system instance in place. (Such imperative edits are brittle because the correctness of the system depends on the correctness of all of these edits over its lifetime.) Since our system instances are immutable, we can refer to them by their secure hashes.

Because of our all-at-once approach to installation, the order in which system instances are produced and installed does not matter, and no sequence of installs and uninstalls can result in a badly formed system instance.

4.5. Runtime

When a system instance boots, the hardware can check that it is the current system instance, and refuse to proceed if it is not.

As stated in Section 4.1, system instances and program instances export values, which can reference other instances; in our example, a P might export two values: a Bool and a K.

```
> data P = P Bool K
```

We let each program read its own program instance at runtime, allowing it to read and act upon the values that it exports. In this case, the Bool might have been a parameter to pModel, intended to control P's execution.

4.6. Policies

Configuring real systems requires one to know a great many *ad hoc* rules. One rule might be that program P is known to work with K and not K′; another might be that

P has not been tested against K″ but that it ought to work anyway—assuming that its `Bool` parameter was `True`. We call these *ad hoc rules,* and we argue that ad hoc rules account for much of the difficulty of real system configuration. Our *system policies* therefore provide a way to express a variety of ad hoc rules that can further constrain the acceptable structure of the system. We need these ad hoc rules because our programs are not perfect, and because their most interesting properties are often not discovered until after they are written and deployed. System administration is often messy and unstructured, and system policies let us express these ad hoc rules.

We can implement many of these system policies using additional type rules. Imagine that program E requires a kernel that supports UTF-8. We can encode this policy by saying that its kernel must belong to the `Utf8Support` type class (perhaps among others).

```
> data E
>   = forall k .
>       (Utf8Support k, HasKReboot k)
>       => E Hash ERun k C P
```

Each known kernel type can then be listed as belonging to the `Utf8Support` type class or not. When new determinations are made—perhaps a new kernel is published, or perhaps an old kernel is found not to support UTF-8 to our satisfaction—we can import new definitions and act on them. While we must make these annotations manually, we can check them automatically.

For other policies, when type rules are not so directly applicable—for example, if there is a policy that the system must fit in less than a megabyte of RAM—an ad hoc checker can traverse the system instance and check it against the desired policy.

Some some system policies can be authored by the local system administrator, while may accompany programs from elsewhere, and yet others may come from third parties. The local system administrator can choose to adopt these imported policies or not.

If a system instance does not conform to the governing policies, the evaluator will not produce it and we cannot use it; we must change the model or its parameters for it to become acceptable.

4.7. Attribution

Another ad hoc policy—for example—might be that the local system must provide a good French-language UI. We might redefine a `System`'s program instances as belonging to the type class `Français`.

```
> data System
>   = forall k c p e .
>       (Français k, Français c,
>        Français p, Français e)
>       => System Hash SRun k c p e
```

We can then define our program instances—E, for example—as belonging to `Français`.

```
> instance Français E
```

But who writes this instance definition? What is a "good" French-language UI? Who gets to decide? And how might we check so ill-defined a policy?

Our rule is that the local system administrator makes such decisions, and a local system instance belongs to the type class `Français` if and only if the local administrator says so. The local administrator can of course choose to defer to the program's publisher when appropriate, or to other authorities—perhaps to the Académie Française [11], which could publish its own policies. The earlier Binder security language provides mechanisms for attribution and deferral ("delegation") in a distributed environment [3] that should be useful here too.

Another policy might more realistically insist that the system's component programs not have been named in US-CERT security alerts [16]. Ongoing security alerts arriving at a system could cause the system no longer to meet its policy, perhaps notifying an administrator.

4.8. Extensions

We hope to specify and check a variety of system properties using the approaches described here, and we hope we can extend these approaches to extend the properties we can specify and check.

Our current system instances are static, but we plan also to support *dynamic instances* to model the system's runtime state. A program will be able to read its own dynamic program instance, referencing other dynamic program instances; this could provide a foundation for easily configurable inter-program communications.

Real system state can seem quite complex. This paper was written on a system with 216,141 files and 17,663 folders, but many of its 233,804 ACLs are little more than accidents of history. While there is little chance that these ACLs are all correct—whatever that might mean!—there may be some greater chance that we can write concise policies that can check the ACLs. Perhaps such system state is not as complex as it seems!

Expressing our ad hoc policies as type rules requires a powerful and flexible underlying type system. While Haskell has an excellent type system, one can certainly imagine possible improvements.

Our current approach to system security is restricted to ensuring system integrity. We hope also to address confidentiality in future extensions.

In our current design we avoid the inviting possibility of fixing system configuration problems automatically as they are detected, such as by substituting a better version of a kernel, since doing so currently seems much more error-prone than relying on humans to fix these problems. We expect to revisit this decision later.

5. Feasibility

Is this approach to system configuration feasible? The only sure way to tell for sure is to build it and use it, but we have some intuitions suggesting that it could work.

While earlier efforts at declarative configuration, such as Vesta and the CML2 kernel configuration language [14], have not been widely adopted, they were targeted at programmers who already used and understood the existing configuration tools, and who were therefore disinclined to switch. This may not be a problem with personal systems, where the need for new tools for users and administrators should be more obvious.

Our system models and system policies may be too complex and too difficult to get right. We argue only that they will be smaller, simpler, and more precise than the system instances they produce and check.

Since many people will write submodels, we must create standards to allow their correct interoperation, and ensure that malicious submodels cannot hijack a system. Our current understanding of these problems is inadequate, but it should improve with further experience.

While we have certainly not eliminated the need for system administration, we believe that we have reduced the work involved. A sufficient reduction should allow us to outsource the remaining administration tasks, including detecting, diagnosing, and repairing any problems that otherwise elude us.

Finally, we note that we have based this work in its entirety on the assumption that the complexity of system configuration limits the use and acceptance of personal systems. We have no quantitative evidence to support this assumption, although we do have a growing collection of supporting anecdotes.

References

[1] J. H. M. Dassen, Chuck Stickelman, Susan G. Kleinmann, Sven Rudolph, and Josip Rodin. *The Debian GNU/Linux FAQ chapter 6—Basics of the Debian package management system.* February 2003.

[2] Christian Collberg, John H. Hartman, Sridivya Babu, and Sharath K. Udupa. "SLINKY: Static linking reloaded." *Proceedings of the USENIX 2005 Annual Technical Conference,* Anaheim, California, pp. 309–322, 2005.

[3] John DeTreville, "Binder, a logic-based security language." *Proceedings of the 21st IEEE Symposium on Security and Privacy,* Oakland, California, pp. 105–113, May 2002.

[4] The GHC Team. *The glorious Glasgow Haskell compilation system user's guide.* Version 6.4, March 2005.

[5] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. "The Vesta approach to software configuration management." Compaq Systems Research Center Research Report 168, March 2001.

[6] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. "The Vesta software configuration management system." Compaq Systems Research Center Research Report 177, January 2002.

[7] David A. Holland, William Josephson, Kostas Magoutis, Margo I. Seltzer, Christopher A. Stein, and Ada Lin. "Research issues in no-futz computing." *Proceedings of the 8th Workshop on Hot Topics in Operating Systems,* pp. 106–112, Schoss Elmau, Germany, May 2001.

[8] Galen C. Hunt and James R. Larus. "Singularity design motivation." Microsoft Research Technical Report MSR-TR-2004-105, November 2004.

[9] Galen C. Hunt, James R. Larus, David Tarditi, and Ted Wobber. "Broad new OS research: Challenges and opportunities." *Proceedings of the 10th Workshop on Hot Topics in Operating Systems,* Santa Fe, New Mexico, June 2005.

[10] Butler W. Lampson and Eric E. Schmidt. "Organizing software in a distributed environment." ACM SIGPLAN Notices **18**, 6 (June 1983), pp. 1–13.

[11] Louis [XIII of France]. "Lettres patentes pour l'établissement de l'académie françoise." January 1635.

[12] Microsoft Corporation. *Microsoft Platform SDK: Windows Installer.* November 2004.

[13] Simon Peyton Jones, editor. *Haskell 98 language and libraries: The revised report.* Cambridge University Press, 2003.

[14] Eric S. Raymond. *The CML2 resources page.* February 2002.

[15] Joshua A. Redstone, Michael M. Swift, and Brian N. Bershad. "Using computers to diagnose computer problems." *Proceedings of the 9th Workshop on Hot Topics in Operating Systems,* Lithue, Hawaii, pp. 91–96, May 2003.

[16] United States Computer Emergency Readiness Team (US-CERT), National Cyber Security Division, Department of Homeland Security. *Technical cyber security alerts,* 2005.

[17] 20th Century Fox. *I, Robot.* Motion picture theatrical release, 2004.

[18] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Yuan Chun, Helen J. Wang, and Zheng Zhang, "STRIDER: A black-box, state-based approach to change and configuration management and support." *Proceedings of the 17th Large Installation System Administration Conference,* San Diego, California, pp. 159–172, October 2003.