Declaring Victory in a Declarative Datacenter: Verification and Transferring Confidence

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract *Operators may appreciate and adopt declarative approaches to defining datacenters, but they will still need sophisticated tools to locate weaknesses, identify hot-spots, and catch errors. Just as usefully, they need means to transfer their confidence from one version of the system to the next. I outline some of these challenges along with our preliminary work in this direction.*

Languages If the datacenter (or datacentre, even) is the new computer [6], what is the language for configuring it? The lure of declarative languages is that they turn low-level, ad hoc languages into targets rather than sources. Just elevating the meaning of existing operations helps: Mahajan, et al. [3] point out that adopting a transactional semantics for configuration changes would have avoided a significant portion of misconfiguration errors. But declarative specifications will not eliminate errors, only move them higher up the chain of abstraction. For instance, Anderson and Scobie [1] say their current system supports "over 2000 parameters" ranging from hardware to access-control. What are the odds operators will get their specifications right?

Analyses In their study of Internet service failures, Oppenheimer, et al. [5] implicitly observe the value in three kinds of tools (the names given to these categories are ours). It is instructive to consider the impact of declarative specifications on each of these:

user-defined property preservation The first is to have high-level properties of desired behavior, and ensure that individual cofigurations match these desired global properties. While this remains a problem in "legacy" (i.e., current) systems, if individual configurations are generated directly from a declarative specification, this problem effectively disappears (or is subsumed in a proof that the generator preserves the semantics of the specification). Bravo!

smell tests The second is to check configurations against well-known properties. These would include both desirable properties (those found in systems that function well) and undesirable ones (those known to lead to faulty behavior). In a higher-level language specification, these would most probably correspond to types or other static analyses. Another win.

Let us pause momentarily to dig deeper.

Smell-tests are useful but fraught with difficulty. In a domain as messy as datacenter configuration, the clean logical niceties of "soundness" and "completeness" seem unattainable. In the absence of robust properties, the best we must hope for is that analyses will find situations frequently enough to be useful but infrequently enough to not annoy. Designing such analyses—and their corresponding user-interfaces—is daunting.

The other validation scenario is not as watertight as it seems, either. When a datacenter configuration has thousands of parameters, operators would benefit from analyses that probe their specification instead of simply accepting it at face-value. Unfortunately, if smell-tests are difficult to design, obtaining properties seems even harder. The essence of any verification process is obtaining a *redundant* statement of the system's desired behavior. In our experience in a more limited domain (access-control), when the specification language is sufficiently declarative, users have great difficulty providing a duplicate statement of behavior.

The heart of this problem is obtaining a redundant specification. If a redundant one isn't available, is there any other?

Yes, indeed, except it describes a different system: the previous version of this one. In reality, the typical operator often has a configuration that is known to "work"; now he has a new configuration that represents a desired change, either in response to new features or, sometimes, as a result of an emergency (as when a security leak is identified). As Anderson and Scobie say [op. cit.], "Small configuration changes also occur very frequently in a complex environment". What the operator really needs to know is, *Will something break if I make this change?* That is, he cares not about correctness of the new system (which is too complex to comprehend authoritatively anyway), but rather about *transferring confidence*: i.e., how to gain confidence in the new version relative to the confidence placed in the old.

This idea is loosely manifest in Oppenheimer, et al. [op. cit.]:

change-impact analysis The third analysis is generalize their suggestion to "help operators understand [...] how their changes to one component's configuration will affect the service as a whole". Here, a declarative language has direct value: information that had to be reconstucted from lower-level specifications is now expressed directly, making the analysis richer, more tractable, and have fewer false-positives and/or false-negatives.

The Margrave Tool Computer systems of a scale that can benefit from a datacenter teem with policies that govern their behavior. These include:

- A configuration model with context-sensitive rules to determine what components can, should, and can't be combined.
- Access-control policies for the data in the datacenter.
- Firewall policies to determine which sub-networks may and may not communicate with each other.
- User-defined routing policies to administer network traffic for specific needs (e.g., QoS).
- Confidentiality and integrity requirements in systems such as Security-Enhanced Linux (SELinux).
- Hypervisor rules that describe desirable and unacceptable interoperation between compartments.

These policies are, in turn, often modular and even distributed.

Many of these uses have evolved their own domain-specific notation for expressing policies, thus enabling rapid system evolution along a critical vector. Some of these apply so broadly that they have evolved into industrial standards, such as XACML for access control. In general, these languages can be treated quite uniformly using standard languages such as first-order logic.

For several years we have been building Margrave [2], an analysis suite for policy languages. Margrave began as a tool specifically for XACML, but is evolving to support languages described more generally. Margrave has two components:

1. A *verification* engine for checking policies against formal properties. This is general enough to encompass both user-defined properties and smell tests. In the access-control domain, for instance, the useful smell tests describe standard domain metaphors such as least-privilege, conflict-of-interest, and separation-of-duty. These need not necessarily hold in a given policy, but the designer may to ensure that they are violated intentionally.
2. More usefully, Margrave offers *change-impact analysis* as a property-free analysis. I will focus on this aspect of the tool in the rest of this document.

Margrave presents changes as the set of inputs that yield a difference in output and, for each input, the corresponding output change. This is an especially concrete representation that users can immediately comprehend, matching a cognitive model called *surprise-explain-reward* [7].

In practice this change can be quite large, so the user needs tools to distinguish the important from the irrelevant. Margrave therefore enables the user to probe this output. For example:

- Is the new policy equivalent to the old one?
- What are the changes when restricted to a certain type? (For instance, restricting attention to those data formerly denied but now permitted access can identify inadvertent leakage.)
- What are the "hot-spots", e.g., one rule change that altered the effect of a large percentage of outcomes?
- What roles or resources had their permissions changed?

While some of these questions are specific to the access-control domain, others are universal to policy analysis.

The trained computer scientist will, of course, recognize that these correspond to *queries* and *views* over the changes, but the operator doesn't need to understand these issues. Even more intriguingly, an operator can ask:

- Confirm that role X did not gain privileges as a result of an edit.

This is, of course, a form of *verification*. But didn't we say operators have trouble expressing properties? In fact, what we've found from talking with users is that, while people often have difficulty expressing properties of a *system*, they have much less trouble stating properties of *changes*: at the very least, they can state the framing conditions that they expect to hold of their edit.

Change-impact analysis is, therefore, a particularly engaging application of formal methods. It weds the benefits of formality—soundness, coverage of a complex state space, etc.—to an informal, inquisitive usage style. The choice of representation of output particularly helps, because operators can more readily work with concrete, extensional representations than intensional ones. This analysis modality has many different "what-if" uses:

Upgrade checking "If I make this change, what will break?"

Upgrade choosing "Will these two different ways of altering the policy yield the same result?"

Refactoring checking "I intended only to improve the structure of my policy; did I accidentally change anything?"

Some users have even adopted Margrave as an oracle for mutation testing [4].

Looking Ahead Of course, this is only a beginning. There are numerous ways in which Margrave must grow to accommodate the needs of datacenters, such as:

- Analyzing these languages independently is one thing, but combining their analysis is an entirely separate challenge, and one we have not confronted. On the other hand, perhaps we won't need to if declarative languages for datacenters take off. Therefore, I view great potential for the synergistic development of such languages with these analyses.
- These languages operate in a very stateful environment. While we have done theoretical work to extend Margrave to accommodate state, I believe the more fruitful direction would be to run the process backwards: from the policies, derive monitors on the state that warn when changes are about to occur.
- The worlds of numbers and of symbols are currently distinct. We would like to integrate the logical power of these languages with the hard numbers we obtain from dynamic analyses of behavior to provide better analysis and prediction.

Ultimately, Oppenheimer, et al. [op. cit.] point to the predominance of misconfiguration errors and say about improving operator interfaces, "This does not mean a simple GUI wrapper around existing per-component command-line configuration mechanisms—we need fundamental advances in tools to help operators understand existing system configuration and component dependencies, and how their changes to one component's configuration will affect the service as a whole". We couldn't agree more: only thus armed can the operator declare victory on the problem of managing the datacenter. Margrave is our step in this direction.

Acknowledgments

I thank Don Batory, Dan Dougherty, Kathi Fisler, Leo Meyerovich, and Michael Tschantz. The ideas described here (only the good ones, that is) are the fruit of our collaborations. I appreciate support from the US National Science Foundation, Cisco, and Google.

References

[1] P. Anderson and A. Scobie. LCFG: The next generation. In *UKUUG Winter conference*, 2002.

[2] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *International Conference on Software Engineering*, pages 196–205, May 2005.

[3] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. *ACM SIGCOMM Computer Communication Review*, 32(4):3–16, 2002.

[4] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *International World Wide Web Conference*, pages 667–676, 2007.

[5] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Usenix Symposium on Internet Technologies and Systems*, 2003.

[6] D. A. Patterson. Technical perspective: the data center is the computer. *Communications of the ACM*, 51(1):105–105, 2008.

[7] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel. Harnessing curiosity to increase correctness in end-user programming. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 305–312, 2003.