Putting the User in Usable Verification

Kathi Fisler
WPI Department of Computer Science
kfisler@cs.wpi.edu
(joint work with Shriram Krishnamurthi, Brown University)

November 3, 2010

A model of usable verification must account for how people interact conceptually with analysis methodologies and tools. End-users do not particularly care about verification. If anything, they care about fostering (partial) confidence in systems they use (though many would prefer systems to "just work"). Verification is but one route to developing confidence in a system. As such, usable verification must meet users on their own terms, with a high payoff-to-effort ratio.

Consider the common model of static verification that consumes a system description and a set of desired properties stated in some formal language. From these, verification reports a counter-example iff the system disagrees with a property. The following type signature captures this model:

$$\textit{System Descr} \times \textit{Properties} \rightarrow \textit{true or counter-example}$$

The result (*true or counter-example*) provides an example (if available) of how the system fails the property. In theory, a user refines either the system description or the property in response to a counter-example, then re-runs the analysis.

We assess the usability of this model by instantiating it for a context in which mainstream end-users develop formal artifacts: defining access-control policies. Modern web-based applications—such as social networks, electronic health records, and office-productivity tools—have made us all into policy authors. From organizing friends lists to creating groups to share documents, applications ask users to classify others into roles and associate those roles with privileges on information. This task is not innocuous: people play multiple roles within organizations or human circles (such as a friend who is also a colleague). Overlapping roles yield subtle access-control decisions. These decisions are sufficiently complex that users often (intentionally or accidentally) create over-permissive policies.

The system description (the artifact to verify) in this case is the user's access-control policy. Policies are sufficient for checking many interesting properties, but sometimes we need to reason about the system resolving from the interaction of policies and the applications they configure. In such cases, we can include application details in the verification model through the usual mechanism of an *environment*.

$$(\textit{Env Model} \times \textit{System Descr}) \times \textit{Properties} \rightarrow \textit{true or counter-example}$$

The application may seem an unusual environment model, as it is typically much more complex than the policy. A user, however, neither knows nor cares about how the underlying application works. In addition, a user is in no position to supply such a model. The policy is therefore the natural artifact subject to analysis. Throughout this position statement, we assume that an artifact to be verified may be augmented with an environment model that may or may not be provided by the end-user of verification.

In terms of properties, end-user policy-authors typically share two high-level concerns: their data should stay restricted within appropriate groups of people (e.g., in social networks), or the data should get to everyone who needs it (e.g., with workplace-based collaboration tools). Such properties tend to take forms like "my doctor can view my health records". This seems an odd property to verify: it shares a form with

policy rules, and is likely already explicitly stated in the policy. A user would be justified in finding this statement (and its verification) redundant, even though subtle semantic relationships between policy rules may cause the resulting permissions to seemingly contradict individual rules.

The challenge of authoring access-control properties is not just another instance of users not being able to write properties. Many situations in which users configure applications necessarily use simple, declarative languages in which the users' statements closely resemble the most evident properties. In this context, we shouldn't be asking users to write properties at all! Rather, we should seek verification models that automatically help them identify problems arising from their configurations.

Generalizing Artifacts and Outputs

In the standard model of verification, properties describe a ground truth against which to analyze a system description. Properties are not, however, the only reasonable representation of ground truth. Consider a situation in which we have a trusted artifact (such as an existing policy) that we need to modify to adapt to a new requirement or to fix a minor bug. In this case, the existing artifact largely represents ground truth, sans the intended effects of the edit. Verification could therefore compare the two artifacts and report on the different behaviors they admit.

A generalization of our model of verification accommodates this perspective. Verification should consume two artifacts, one of which captures some notion of ground truth, and produce the set of behaviors of the given artifact that are not shared with the ground-truth artifact:

$$\textit{New Artifact} \times \textit{Ground Truth Artifact} \rightarrow \textit{Difference}$$

The generalization of the output to a set of differing behaviors covers the standard output of a counter-example (which is just some element of the set), but is necessarily richer. Some difference in behavior between the two artifacts is expected, otherwise the edit that yielded the new artifact would have achieved nothing. This in turn implies that not all differences represent problems. By returning the set of all differences, we enable tools to help the end-user separate the differences into expected and unexpected ones.

We use the term *change-impact analysis* for an instance of this model in which both artifacts are versions of an evolving system description. Change-impact analysis avoids the common verification challenges of writing properties and determining whether one has enough properties. Under the standard model, a user might miss a crucial impact of an edit if no stated property checked for it. Since our revised verification model produces the set of all differences, the user gets concrete instances of all changes without having to seed the process with properties. Other problems that reduce to comparing artifacts, such as checking whether a refactorization of an existing artifact preserves the original semantics, fit this model as well.

Although change-impact is a property-free analysis, properties still have a useful role in checking system edits. Imagine that a social-network user edits his configuration to grant access to new info to his high-school friends; however, he does not want the changes to leak new information to any of his co-workers. This check constitutes a property on the edit, but it is not a property on either the previous or the new configuration (co-workers have access to some of the user's information in both versions). Checking the consequence of an edit requires that the set of differences resulting from verification itself be a verifiable artifact. Note also that while general properties of a policy might be hard to envision, properties of edits are easier to conceptualize, since the user had some goal in mind when making the edit.

This example illustrates that we can better accommodate users in verification by generalizing its inputs and making its outputs comprehensive and verifiable. Exhaustive, analyzable outputs are particularly important in cases where the end-user doesn't have access to a system description to help in making edits. For example, if a social network changes its application to use its members' configurations differently, a user might want a comprehensive report on the impact of that change. Change-impact analysis could accommodate that by treating the combination of their new application and the user's policy as the artifact to verify, and the combination of their old application and the user's policy as the ground-truth artifact. Models that help users with only partial system descriptions are another key component of usable verification.

Accounting for Users' Perspectives

Our previous discussion about change-impact does not imply that end-users never have properties to verify. Users do sometimes have properties, but framed in different terms that appear in the system model. For example, people often express desired privacy or access settings in terms of how much they trust others (i.e., "I trust professors more than students, so professors should have more privileges"). Trust is omitted from system descriptions with good reason: it is subjective. How can verification assist users in assessing subjective system descriptions?

Once again, we must revise our model of verification. In addition to an objective system description and a notion of ground truth (that may reference a user's subjective perspective), we need an additional model of the user's perspective that references data in the system description. To accommodate our trust property, for example, a user could define a moreTrusted relation on roles or users (which are in the system description). The property could be stated in terms of moreTrusted, while the system description provides the ascribed privileges against which to check the property.

$$\textit{New Artifact} \times \textit{Ground Truth Artifact} \times \textit{User Perspective} \rightarrow \textit{Difference}$$

The user-perspective model is not an environment model: it neither controls nor influences variables in the system description. Rather, it captures attributes of system information (such as moreTrusted) that are relevant to the user in working with or assessing the system, but are not relevant to the system's operation.

From Verification to Authoring

We argued earlier that usable verification needs to assist users who don't care about it in its own right. This suggests making verification part of a development process, rather than a separate task that authors have to execute explicitly. In the case of access-control policies, this suggests that verification should be integrated into policy authoring tools, running automatically in the background to warn users of problems or conflicts that their edits have created.

In the access-control domain, integrating verification and authoring impacts both processes. End-users provide two artifacts to verification: the policy to analyze and (possibly) data that comprises their perspective model. Each of these can evolve throughout the authoring tasks. Verification can check user-defined properties (if any), system-defined properties (such as no conflicting decisions within the policy), or summarize change-impacts after each editing step. This rich integration provides users with the benefits of verification in a light-weight manner that they can exploit or ignore as they see fit.

Summary

This position statement envisions "usable verification" as a set of practices that go far beyond tool interfaces, domain-specific analyzers, or property templates that simplify the task. Advances in these areas are no doubt important, but overlook fundamental questions about how users gain—and maintain—confidence in systems that they only partially control. In particular,

> *Usable verification helps those who develop or merely configure systems to gain or maintain confidence in those systems, even when confidence is based on subjective or external attributes of system data.*

In thinking about usable verification over the years, we have shifted to viewing verification as a means to *transfer*, as well as *establish*, confidence in a system. The change-impact modality targets confidence transfer. We also view our revised model as shifting from intensional behavioral representations based on properties to extensional representations based on concrete differences. Both shifts strike us as key ingredients to achieving usable verification tools.

Naturally, usable verification has many open problems. A sample of these include:

- *How can we make differencing computationally effective for state-based verification?* We have used differencing effectively on declarative models. In this setting, differences are tractable to compute and easy to present. Once differences consist of sets of traces, they become both harder to compute and harder to present.

- *How do we effectively present large sets of differences to users?* Even simple edits can yield dozens if not hundreds of concrete differences. Just as some verification work has sought to identify effectively similar counter-examples, we need techniques to identify the most interesting or unique differences.

- *What are effective metrics for evaluating usable verification tools?* One test of whether we have a clear model of usable verification is whether we also have metrics for deciding whether a verification approach is indeed usable. We've been using Cognitive Dimensions analysis in some of our work, but that is geared towards evaluating techniques that target concrete tasks rather than verification's fuzzier goal of helping someone gain confidence.

- *How can we help users identify and develop good models of relevant perspectives?* Our notion of perspective models arose from observations about how users think through decisions during policy authoring. Expecting users to develop both perspective models and properties against them during routine system configuration, however, is a bit unrealistic. While we suspect such models are indeed a key component of usable verification, we have to figure out how to not create additional usability problems while supporting these models.