Evidence Management in Programatica

Mark P. Jones

[1] OGI School of Science & Engineering at OHSU
[2] Portland State University

This paper summarizes our efforts in the Programatica project at OGI and PSU to design new kinds of tools to support the development and certification of software systems. Our approach relies on a tight integration of program source code, embedded formal properties, and associated evidence of validity. A particular goal for the toolset is to facilitate efficient and effective use of many different kinds of evidence during project development. Our current prototype targets validation of functional (security) properties of programs written in Haskell. This tool provides connections, through a language of formal properties called P-logic, to several external validation tools and supports unit testing, random testing, automated and interactive theorem proving, and signed assertions. The underlying concepts, however, are quite general and should be easily adaptable to other programming languages and development tools, and to support a wide range of both process- and artifact-oriented based validation techniques.

1 Software Development and Evidence Management

Software developers rely on a wide variety of techniques to assure themselves (and their customers) that the system they are building will function correctly:

- In the early stages of a project, experiments, models, and prototypes can be used to gain a better understanding of system requirements, and unit tests or specific test data sets may be collected to document expected behavior.
- Developers might also use tools based on static analysis or formal methods—such as model checkers or theorem provers—to obtain evidence for key properties. Tools like these can provide strong guarantees about program behavior that are particularly important in safety or security critical applications where high levels of assurance are required. However, effective use of such tools can require significant investment in both initial training and daily use, and hence they are often considered too expensive to be used in the early, more fluid stages of project development.
- Process-oriented techniques—including, for example, code inspection and design review—are an important component of current certification methodologies and can often be used at multiple stages of the development process.

There are of course, many other techniques that could be listed with the above examples, some of which are quite general while others are applicable only in specific domains. But although there are many different techniques, there is still one important common feature: each of them results in some tangible form of *evidence* about specific properties of the software system. For example, an input

and (expected) output pair can be used to document a test case; a script or tactic can be used to record the structure of a formal proof; and detailed minutes can be used to capture the results of a code review meeting.

Some of the biggest challenges for anyone tasked with certifying the behavior or properties of a software system are in managing, maintaining, and exploiting the large and diverse volumes of evidence that are created during its development. This observation motivates the development of new tools and techniques that will allow evidence to be reused, repeated, or replayed so that validity of each component in a system can be monitored automatically and incrementally without the need to reconstruct evidence from scratch at every step, or when the development is complete.

2 The Programatica Approach to Evidence Management

Our specific goals in the Programatica project are to build tools that allow users: to capture evidence and collate it with source materials; to exploit dependencies between evidence and the programs to which it refers as a means of tracking change; to automate the process of combining and reusing evidence; and, finally, to understand, manage, and guide further development and validation efforts. In addition, we recognize that evidence may come in many different forms, and that tools must be designed to address this: a key feature of our approach is the use of a general *certificate* abstraction for encapsulating, accessing, and manipulating different forms of evidence in a uniform manner.

Our approach is intended to be quite general, but it is inspired, in part, by techniques adopted in more specialized tools. The practice of "Extreme Programming" [2], for example, encourages frequent use of testing as an integral part of coding and refactoring [4] and has stimulated the development of tools that automate the testing process. These tools, however, do not attempt to deal with or incorporate other kinds of evidence. Similarly, compilation tools (such as `make` [3]) track dependencies between source code units to minimize the need for recompilation, but they do not attempt to capture other kinds of dependencies or evidence. As a final example, some systems support "external oracles" that allow users to integrate theorem proving with other validation tools such as BDDs [5] or model checking [1]. These tools, however, focus on formal validation and do not directly address evidence capture and management.

2.1 Certificates

Programatica certificates are a mechanism for encapsulating different types of evidence. The evidence itself, as well as the internal format by which it is represented, will vary from one certificate to the next. But, from the perspective of an evidence management tool, every certificate offers the same basic interface, with attributes that describe its *sequent* and *validity*, and operations that allow certificates to be *validated* and *edited*. Each of these features is described below:

– The *sequent* of a certificate formalizes the claim that the accompanying evidence is intended to support. Sequents provide the means by which disparate kinds of evidence can be brought together in a single environment. In our current system, we write sequents as judgments, $\Gamma \vdash \Gamma'$, where the *hypotheses* in Γ and the *conclusions* in Γ' are lists of formulas over a suitably chosen specification logic. The formulas in a sequent may include direct references to variables and functions that are defined in the source text. As such, a sequent also provides a starting point for tracking dependencies between certificates and the underlying code base.
– A certificate is *valid* if its sequent is consistent with the evidence it contains. For example, a certificate with sequent $\vdash A$ is valid only if it provides evidence for A. In this way, validity serves as a contract between external tools and the evidence management system.
– The actions needed to *validate* a given certificate will depend on the type of the certificate, and may, in some cases, involve significant computation. To permit a quick test of validity, each certificate includes a flag that is set only when the certificate is known to be valid. If either the certificate itself or a part of the source text that it depends on is changed, then the flag will be reset and the full test of validity can be deferred until needed.
– The actions needed to *edit* a certificate—such as modifying it so that its validity can be established—will also depend on the type of the certificate, and may, in some cases require significant user interaction.

The Programatica certificate abstraction supports compositional certification. For example, from a certificate with sequent $A \vdash B$ and another with sequent $B \vdash C$, we can derive a compound certificate with sequent $A \vdash C$. If changes to the underlying program invalidate only one of the original certificates, then we will not need to construct new evidence for the other one or for the composite. Note also that the original two certificates could be constructed using different tools; the composite can then be tagged to reflect the set of tools that were used in its construction, and this information can be used as an indication of its pedigree. In this way, certificates and sequents provide a mechanism for integrating and reasoning with different kinds of evidence.

2.2 Certificate Servers

Certificate servers (or just 'servers') play an important role as a mechanism for creating and using different types of certificate in a uniform manner. We distinguish between: *external* servers, which are used for certificates whose evidence is supplied by external tools; and *internal* servers, which use functionality that is built in to the evidence management tool, and provide a means for combining different types of evidence.

– External servers connect the evidence management system to the external tools that are used to construct and maintain evidence. As such, external servers can be understood as software plug-ins that must be installed before certificates of a particular type can be edited and validated. External

servers are responsible for translating between the languages used in source documents and sequents and the languages used by external tools. It is the responsibility of each external server to detect and report cases where translation is not possible. A second responsibility of an external server is to capture and package context from source documents so that it can be used by the external tool. We refer to this as 'theory formation' because it corresponds to assembling a theory that includes the facts and definitions that would be needed to prove a particular theorem.
- Internal servers provide built-in functionality for generating and combining evidence. This includes 'axiom' servers that can generate and validate certificates for sequents of a particular form and 'rule' servers that can be used to combine previously constructed certificates.

Servers provide an infrastructure for theorem proving with certificates in which different servers correspond to different external oracles and inference rules. One of the most tricky design choices here is to determine how much of this machinery should be built in to the evidence management tools, and how much should be delegated to an external theorem prover.

3 Challenges for Future Work

The Programatica approach to evidence management offers a new vision for high-assurance software development and certification. Our current prototypes [6] are in an early stage of development but are designed to extend current evaluation methodologies by supporting and integrating the different kinds of evidence that they require. We hope that Programatica will also provide an evolution path for introducing and applying formal methods to document and validate essential functional properties of critical software systems at high assurance levels.

There are, however, many challenges to address and evaluate with future generations of the Programatica tools, including:

- What can a tool do to help users visualize and understand the evidence they have assembled, to prioritize future validation tasks, and to identify areas in which evidence is either lacking or weak?
- How can we deal with differing levels of trust and confidence in the different kinds of evidence, servers, and models that are used?

Certainly, some aspects of confidence and trust can be quantified. For example, if one test suite includes all of the tests from another, then the first should offer at least the same degree of assurance as the second. But many other aspects are subjective and will require a flexible tool that can be tailored to policies of an organization or to the requirements of a particular certification process.

Acknowledgments

The work described in this paper has benefited significantly from the input of Programatica team members including Thomas Hallgren, James Hook, Dick Kieburtz, Rebekah Leslie, Andrew Tolmach, and Peter White.

References

1. Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-FL: A pragmatic implementation of combined model checking and theorem proving. In *Theorem Proving in Higher Order Logics (TPHOLs)*, July 1999.
2. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
3. S.I. Feldman. Make-A program for maintaining computer programs. *Software—Practice and Experience*, 9(4), 1979.
4. Martin Fowler et al. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1999.
5. Michael J.C. Gordon. Reachability programming in HOL98 using BDDs. In *Theorem Proving in Higher Order Logics (TPHOLs)*, August 2000.
6. The Programatica Team. Programatica tools for certifiable, auditable development of high-assurance systems in haskell. In *High Confidence Software and Systems*, Baltimore, MD, 2003.