Dealing with Evidence:
The Programatica Certificate Abstraction

Mark P. Jones

Department of Computer Science & Engineering
OGI School of Science & Engineering at OHSU
20000 NW Walker Road, Beaverton, Oregon OR 97006, USA
`mpj@cse.ogi.edu`

Abstract. In software projects, developers often rely on a wide variety of evidence to assure themselves that the system they are building is functioning correctly. There are many ways to generate evidence—from code reviews to testing and theorem proving—but the diversity and volume of evidence can be hard to manage, maintain, or exploit as a project evolves and meaningful levels of assurance are required.
In this paper, we describe a new kind of tool that facilitates effective use of evidence throughout a project. Such tools should allow users to capture and collate evidence with source materials; to exploit dependencies; to automate combination and reuse; and to understand, manage, and guide further development and validation efforts. Our work is presented in the context of a prototype built for the Programatica project at OGI where evidence is represented by a *certificate* abstraction, but the key ideas, we believe, should be more widely useful.

1 Introduction

Throughout the lifetime of a software project, the designers and developers will typically gather, generate, and rely on many different kinds of evidence, formal and informal, to assure themselves that the system they are building is functioning correctly. The nature, quality, and quantity of this evidence will often vary depending on factors such as the overall project goals, its stage of development, the needs of programmers and users, and the level of assurance that is required:

- In the early stages, when many questions about the basic design are yet to be resolved, one might expect the developers to assemble evidence that includes descriptions of basic requirements, assumptions about the environment in which the system will be operated, manufacturer specifications for components that might be used, and preliminary design documentation such as rough sketches or simple, proof-of-concept prototypes.
- As the project begins to mature and a body of executable code starts to take shape, evidence such as unit tests, or specific test data sets, may be collected, both to document expected behavior, and to ensure that previously detected and corrected bugs are not inadvertently reintroduced. Detailed assumptions

and intuitions about the way the code works—from high-level architectural perspectives to low-level issues like the selection and implementation of data structures—may be documented at this time, and the evolving system may be subjected to code inspections and reviews. Diagnostics from compilers and program analysis tools may also be used to identify problem areas and construct to-do lists of tasks that require further attention.
- Once the basic structure is firmly in place, developers might use tools based on formal methods—such as model checkers or theorem provers—to establish or obtain evidence for key properties. Effective use of such tools can require significant investment, both in initial training and in daily use, and may, therefore, be considered too expensive in the early stages of a project while significant structural changes are still likely. On the other hand, these tools can also be used to obtain very strong guarantees about program behavior—such as freedom from deadlock, or verification of security or communication protocols—that will be particularly important in safety or security critical applications where a high degree of assurance is required.
- Finally, as the project approaches a release date or is prepared for deployment, more emphasis is likely to be placed on compatibility testing, performance tuning, usability issues, documentation, and installation procedures.

In practice, few (if any) software projects follow such a simple, linear course of development with clear and neat divisions between different stages. Instead, they evolve in complex and unpredictable ways as bugs are detected and fixed, as users clarify or change their needs or request new functionality, and as the development efforts adapt to reflect changing priorities and emphasis. This suggests a new opportunity to use evidence as as a mechanism for identifying and tracking change. Moreover, it suggests that we need new ways to allow evidence to be reused, repeated, or replayed so that validity of an evolving system can be checked without the need to reconstruct evidence from scratch at every step.

1.1 This Paper

This paper summarizes our efforts to design new kinds of tools that facilitate efficient and effective use of evidence throughout a project. More specifically, our goal is 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. For example, each of the following may be useful as evidence:

- An assertion of validity, digitally signed by the person who makes the claim;
- A set of test data, including the expected output and the date when the tests were last run;
- A citation, URL, or the full text of a document that provides the proof of a theorem or the specifications of a component;

- An encapsulated session with some external proof tool containing a partially completed attempt to prove some associated theorem. Such a proof might be completed at a later date by resuming the session with the external tool.

To support such diversity, our tools will need to handle evidence using generic interfaces, and they should also be extensible so that they can accommodate new kinds of evidence as they are needed. There are also some less well-defined issues that our tools will need to address:

- How can we deal with differing levels of trust and confidence in the different kinds of evidence that are used?
- 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?

These are challenging problems. 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 adapt to individual preferences and biases. For example, while some users might be satisfied with proofs from a theorem prover, others more skeptical—perhaps concerned about the possibility of errors or over-simplification in the formal model—may prefer careful and extensive testing.

We are not aware of previous work to build tools for evidence management with the same broad scope we have described here, but there are many more specialized tools from which we can draw inspiration. The practice of "Extreme Programming" [3], for example, encourages frequent use of testing as an integral part of coding and refactoring [9]. These ideas have stimulated the development of tools to automate the testing process, but they do not attempt to deal with or incorporate other kinds of evidence. Similarly, compilation tools (such as `make` [8], or the SML/NJ compilation manager [5]) 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 mix different kinds of evidence by integrating theorem proving with other validation tools such as BDDs [10] or model checking [1]. These tools, however, focus heavily on formal validation and do not directly address evidence capture and management.

Our work to date has been carried out in the context of the Programatica project[1] at OGI, whose goal is to develop a new kind of program development environment that encourages its users in stating, thinking about, and validating key properties of the software they are writing. Programatica augments the functional programming language Haskell [14] with a notation (and an associated logic) for stating (and reasoning about) properties of executable code; it provides mechanisms for exporting these properties in an appropriate form to a variety of external validation tools; and it uses a *certificate* abstraction as a

[1] `http://www.cse.ogi.edu/PacSoft/projects/programatica/`

way to capture evidence of validity. For further context, the screenshot in Fig.1 shows one possible user interface for Programatica, which consists of a main window with two sub-panes. The pane on the left is a tree-based project browser.

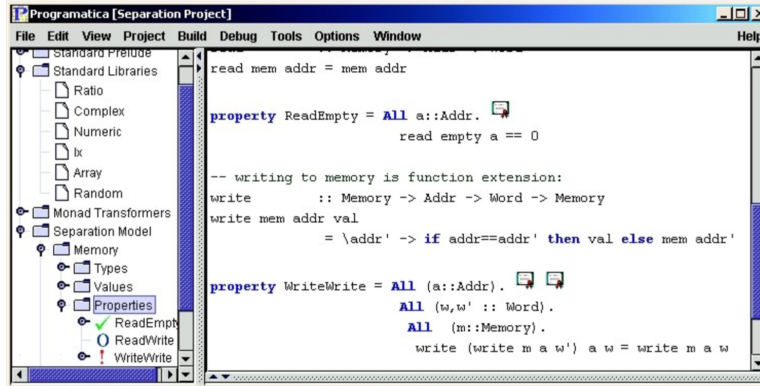

Fig. 1. A mockup of a possible graphical user interface for Programatica.

The pane on the right shows a Programatica source *document*, including both definitions of executable code and assertions of key properties. Certificates of validity have been provided for some of these properties; these are indicated by the presence of small certificate icons that are embedded in the source text. Although Programatica has been designed with Haskell in mind, much of the basic infrastructure could be adapted to other languages. For example, a version of Programatica for Java might be based around the Java Modeling Language (JML) [12, 15] as a notation for expressing properties of Java code and integrating theorem proving via LOOP [16], model checking via Bandera [6], automatic invariant detection via Daikon [7], extended static checking via ESC/Java [13], and other techniques such as runtime assertion generation [4].

In short, as we have worked to develop the foundations for Programatica, it has become clear that the basic certificate mechanisms are essentially orthogonal to other aspects of the design, and that they may well have much wider applicability. We will see this concretely in the general roles that both documents and certificates play in this paper.

The remaining sections of this paper are as follows. In Sec.2 and Sec.3, we describe the use of compound documents and certificates, respectively, in capturing source materials and associated evidence. The role of *certificate servers* in supporting different types of evidence is presented in Sec.4. To explore our ideas in more concrete form, we have built a prototype tool, which has also given us an opportunity to experiment with user interface issues described in Sec.5. A quick overview of the prototype implementation is included in Sec.6. We conclude with a brief summary in Sec.7.

2 Documents

The first challenge of evidence management is in being able to capture many different kinds of evidence, and in collating that evidence with corresponding sections of source code. To meet these needs, we take inspiration from the so-called *compound document* technologies of modern office application suites, which allow users to embed spreadsheets, database tables, charts, and other objects in word processed documents. From the user perspective, each document is stored in a single file that can be copied, renamed, or deleted just like any other file. The internal file format, however, provides a structure more like a hierarchical file system. For example, the main text might be stored in a 'file' in the compound document's root folder, while individual attachments of different types are placed in separate subdirectories.

The diagram in Fig.2 shows how these ideas are adapted to Programatica documents. In this case, each compound document contains a program source

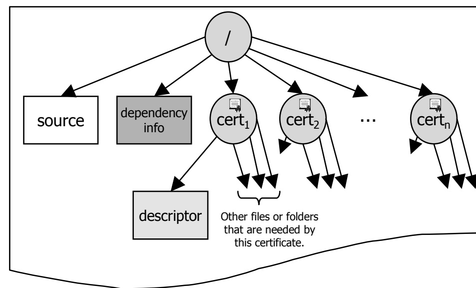

Fig. 2. A Programatica compound source document includes conventional program source text, dependency information, and a collection of certificates.

text; cached information about dependencies between program elements and certificates (to permit faster recompilation and validation); and a collection of subdirectories representing certificates. It is also possible for such compound documents to include image data, media files, property lists, or any other resources that programmers might wish to package up with particular source files.

In this paper, we focus on the folders that are used to store certificates within a document. In particular, we need (i) a way to associate individual certificates with particular points in the program, and (ii) a way to inspect and use the data in each certificate folder in an appropriate manner.

One way to meet need (i) is to use a special form of comment annotation in the source text. In Programatica, for example, we can use a Haskell comment of the form `{-#cert:lemma1#-}` to indicate the intended position in the document of the certificate stored in the folder called `lemma1`. In a GUI editor, like the one in Fig.1, the annotation can be displayed as an icon. However, by avoiding

special syntax or binary representations, the basic program source text will be kept in a form that can also be used by programmers who prefer conventional text editors and standard command line programming tools.

To meet need (ii), we will require each certificate folder to contain a *descriptor* file that uses a standard format to capture important attributes such as the name, type, and status of the certificate. Some of these details can be used to provide a quick description of the certificate, without needing to probe the contents of its folder more deeply. For other tasks—such as opening a certificate for editing or validation—the certificate type must be used to determine how additional files or subfolders should be interpreted.

For flexibility and extensibility, we can adopt a standard technique from component-based programming, representing each different type of certificate by a globally unique identifier, or GUID[2], and using a *registry* to associate individual GUIDs with corresponding *certificate servers*. Thus each type of certificate can have a specialized server program that can access and use any extra data in the corresponding certificate folder.

3 Certificates

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, the most important aspects of which are the attributes that describe a certificate's *sequent* and *validity*, and the operations that allow certificates to be *validated* and *edited*. Each of these features is described in the following subsections.

3.1 Sequents

The sequent of a certificate formalizes the claim that the accompanying evidence is intended to support. More generally, sequents provide the means by which disparate kinds of evidence can be brought together in a single environment. In this paper, we will write sequents in the form of judgments $\Gamma \vdash \Gamma'$, where the *hypotheses* in Γ and the *conclusions* in Γ' are lists of logical formulae over some suitably chosen specification logic[3]. In particular, the formulae in both hypotheses and conclusions may include direct references to entities such as variables and functions that are defined and used in the source text. The intuition

[2] It is common to use strings of 128 bits as GUIDs, and to generate new GUIDs by hashing time, date, and network address information with randomly generated data; the goal is to make it (practically) impossible for independent developers to pick the same GUID for different components.

[3] Our use of the term 'sequent' is consistent with its use in logic, and with the implementation used in our current prototype. It is, however, more specific than we really need for a general evidence management system, and there are other forms of sequent that could be used instead.

for a sequent $\Gamma \vdash \Gamma'$ is that one or more of the formulae in Γ' can be guaranteed to hold when all of the formulae in Γ are satisfied. Thus sequents may be used to state both facts with an empty set of hypotheses (for example, limits on the possible values of a sensor reading, which might influence the selection of a particular representation), and conditional statements with a non-empty set of hypotheses. The empty sequent, denoted $\vdash$, can also be used for some types of certificate if the claim that they might support is judged to be either too informal or too specific to be reflected as a formal judgment.

The task of choosing a suitable logic for the formulae in certificates may not be easy, and will depend on context. For instance, in Programatica, we are experimenting with a logic of partial functions, while for Java we might adopt the JML logic [15]. In our prototype, we have so far avoided this issue by allowing only atoms (i.e., basic propositional variables) as the formulae in sequents.

3.2 Validation and Editing

A certificate is *valid* if its sequent is consistent with the evidence it provides. For example, a certificate with sequent $\vdash A$ is valid if it provides evidence for A, but invalid if it contains either incomplete evidence or evidence for a different formula B. In the latter case, there are at least two ways to make the certificate valid, either by changing its evidence to support A, or by changing its sequent to $\vdash B$. In this way, validity provides an interface between the evidence from external tools and the language of sequents that is used for evidence management.

The actions needed to determine whether a given certificate is valid will depend on the type of the certificate, and may, in some cases, involve significant computation. (Section 4 includes specific examples of the steps that are needed to validate the certificate types used in our prototype.) To permit a quick test of validity, each certificate includes a flag that is set to true 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 set to false. This records the fact that a subsequent validity check is required, and does not necessarily mean that the certificate is, in fact, invalid.

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 screenshots in Fig.3 show the editors for three of the different kinds of certificate in our prototype. Clearly, there is some commonality in the editors—each of dialog boxes shows an icon and a name for a particular type of certificate, and an indication of the given certificate's status (either valid or unknown). At the same time, there are also some significant differences from one certificate type to the next. The leftmost editor, for example, is for automated testing, and does not display a sequent because all certificates of that type use the empty sequent, $\vdash$. The rightmost editor is used with certificates obtained by resolution (See Sec.4.2) and includes several fields that are not present in the other examples. Where appropriate, the editor dialog for a certificate includes buttons allowing the user to invoke associated external tools with appropriate settings for that certificate and hence

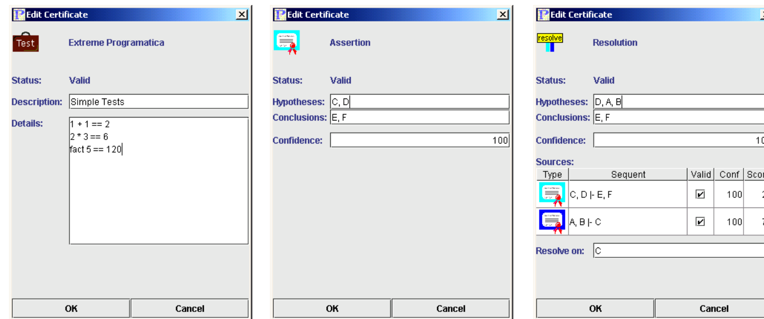

Fig. 3. Examples of the certificate editors in our prototype. The three editors shown are for automated unit testing (left); assertions (center); and resolution (right).

to work with and modify the underlying evidence. (None of the examples in the figure require this functionality.)

4 Certificate Servers

Certificate servers (or just 'servers') play an important role as the primary mechanism for creating and using different types of certificate. For example, once an appropriate server has been located, we can use it to reconstruct the certificate that is stored in a particular folder of a compound document. Servers are responsible both for creating appropriate certificates, and for endowing them with the functionality that is needed for validation and editing.

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. These two kinds of server are described in Sec.4.1 and Sec.4.2, respectively.

4.1 External Servers

External servers connect the evidence management system to the external tools that are used to construct and maintain evidence. As such, external servers will perhaps be most visible to users as the 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. In some cases, there will, inevitably, be mismatches that cannot easily be bridged. For example, an external satisfaction checker that provides a decision procedure for first-order formulae with quantifiers over booleans only will not accommodate arbitrary formulae from an evidence management system's high-order predicate logic. This does not mean, however, that the two tools cannot be used together,

or that the evidence management system's choice of logic must be reduced to some 'least common denominator' of all the external tools to which it may be connected. Instead, we make it the responsibility of the external server to detect cases where translation is not possible. Amongst other things, this motivates the use of a rich and expressive internal logic that can accommodate the logics of many externals tools (or, at least, substantial portions of those logics). It also suggests that different external tools are described and cataloged carefully so that users can be quickly guided to an alternative when their first choice of an external tool proves to be unsuitable. Note also that translation is not a one way process. For example, to achieve a high level of integration, the counterexamples that a model checker produces when an asserted condition fails should, ideally, be translated back to use the same notation in which that assertion was expressed.

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. In the context of an external theorem prover, for example, we refer to this as 'theory formation' because it will require assembling a theory that includes the facts and definitions that are needed to prove a particular theorem.

Translation and theory formation are challenging tasks, and neither one has been addressed by the (almost trivial) external servers in our prototype. We expect this to be an important area for future work, but note also that some significant progress here has already been made in the Programatica project in developing interfaces to both HOL98 [2] and Alfa [11].

4.2 Internal Servers

Internal servers do not require specific external tools, and so provide built-in functionality for generating and combining evidence. In this section, we describe some of the internal servers in our prototype; this is intended as an indication of the kinds of functionality that can be supported, and not necessarily as examples of servers that would be included in a full evidence management system.

"Axiom" Servers The simplest kinds of internal servers can directly generate and validate certificates for sequents of a particular form, and are analogous to axioms in a logical system. There are two examples of this in our prototype:

- **Trivial Sequents**: The trivial sequent server produces and validates certificates for sequents of the form $\Gamma \vdash \Gamma$. The server requests an initial value for Γ when it is used to create a certificate of this type. Subsequently, the left and right hand sides of the sequent may be edited independently. This, however, will invalidate the certificate, and the server will not allow it to be revalidated unless the two sides are equal, which may require further edits.
- **Monotonicity**: The monotonicity server can be used to obtain certificates with sequents of the form $\Gamma \vdash \Gamma'$, where $\Gamma' \subseteq \Gamma$. The current implementation of this server does not prompt the user for Γ or Γ' when a new certificate is created, but instead defaults to the empty sequent, $\vdash$, which is valid, but

not particularly useful. Subsequent edits can be used to set different values for Γ and Γ'; the modified certificate can be revalidated whenever $\Gamma' \subseteq \Gamma$.

Clearly, the second of these is strictly more powerful than the first because every trivial sequent can be established using monotonicity. Nevertheless, in a practical system, when the costs of validation are taken into account, it may be very useful to have both weak but efficient servers to deal with easy special cases, and more powerful but also more expensive servers to deal with harder, general cases.

"Rule" Servers Other types of internal server rely on the results of previously constructed certificates. Servers like this correspond to rules in logical systems and provide the key mechanism by which individual pieces of evidence are combined. In general, when a certificate c is constructed by making use of a previously constructed certificate c', we refer to c as a *client* of c'. Of course the resulting dependencies between certificates must be taken into account in determining validity. For example, if the user changes, and hence invalidates a certificate c', then every client c should also be invalidated. Notice that, by ensuring each certificate is constructed before its clients, we can at least be sure that there are no circular dependencies.

Examples of "rule" servers in the prototype include:

- **Copy**: The copy server will construct a certificate c by using another, previously constructed certificate c', and will initialize the sequent for c with a copy of the sequent for c'. The sequents associated with either or both of these certificates may be modified by subsequent edits, but this will invalidate c (at the very least), and the copy server will not allow it to be revalidated if c' is invalid or if the sequents of c and c' differ.
- **Weakening**: The weakening server supports the construction of certificates c in which the sequent is obtained by weakening the sequent of another certificate c' (i.e., by adding extra hypotheses on either side of the sequent).
- **Resolution**: The resolution server is the most sophisticated internal server in the prototype, and is based on resolution (or, to be more precise, on the 'cut' rule of sequent calculus; unification plays no part in the prototype's underlying propositional logic):

$$\frac{A, X \vdash B \quad C \vdash X, D}{A, C \vdash B, D}$$

  Here, A, B, C, and D are arbitrary sequences of formulae and we emphasize the special role that X plays as a part of both hypotheses, by referring to the rule as "resolution on X." When the resolution server is invoked, it prompts the user to enter a value for X, and then searches for ways to apply resolution on X to the sequents of the currently selected certificates. If resolution cannot be used, no certificate is constructed and an error diagnostic is displayed. On the other hand, if there is more than one way to apply resolution, then the user is presented with a list of possible results and asked to select one of them. If there is only one possible choice, then the server just constructs the necessary certificate and inserts it into the host document.

Thinking beyond these specific examples, it is clear that internal servers provide the infrastructure for interactive theorem proving, with different servers corresponding to inference rules or, depending on your perspective, tactics. In a practical system, with a richer underlying logic, it would be useful to include more powerful internal servers to automate and combine primitive tasks including, for example, quantifier elimination, matching and unification, simplification and rewriting. Adding a degree of programmability would give further flexibility, allowing users to develop and use custom libraries of derived rules and tactics.

5 User Interface Issues

One of the most challenging practical aspects of building an evidence management system is in developing an interface that will help users to work more effectively and to understand the details of a complex project more easily. In this section, we describe some of the ideas for visualization of documents, servers and certificates that we have been experimenting with to address these needs in the context of our prototype. Several of these ideas are illustrated in Fig.4.

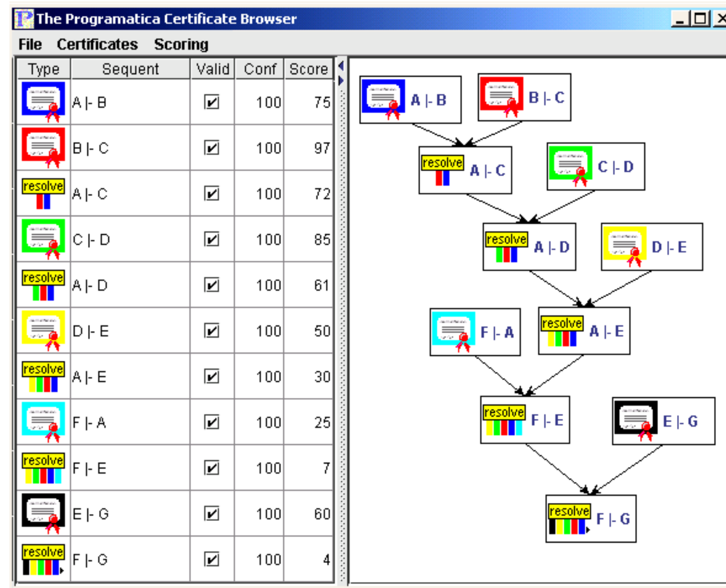

Fig. 4. A screenshot from the prototype showing two views of a particular document. On the left is a table that lists the certificates in the order they were created. On the right is a graphical display that highlights dependencies between certificates.

As the figure shows, every certificate is represented visually by an icon. The standard icon for certificates obtained from an external server is a conventional

certificate with a border color that can be used to distinguish between different servers. Useful information can be conveyed in the choice of colors: for example, less reliable forms of evidence might be displayed with red borders, alerting users to the possibility that stronger evidence might be needed.

The icons for the certificates of internal servers are annotated with colored “ribbons” that represent the external servers on which the certificate depends. As a result, the red coloring from an unreliable external certificate will propagate to each of its clients. Of course there is a limit to the number of different ‘ribbons’ that can be displayed within an icon, and there are also some interesting questions about how we might use other visual attributes, such as the ordering or width of ribbons, to best reflect the quality of the underlying evidence.

Early experiments with the prototype suggested that visualization of dependencies between certificates would be useful. This resulted in the development of the second (right-most) document view in the figure, which shows these dependencies in a simple and intuitive way. A lingering concern is that this graphical view will become harder to work with as the number of certificates increases.

A final comment is required to explain the references to confidence levels and scores that some readers will have spotted in our screenshots. These are a sign of the pragmatically motivated experiments that we have been using to evaluate various schemes for comparing the quality of evidence in individual certificates, and for establishing priorities, based on user specified preferences, that can guide further validation efforts. At present, user input is provided by assigning ‘confidence’ levels to servers and to individual certificates, and by selecting between different methods for calculating ‘scores’. For example, one such algorithm calculates the score of a certificate as the minimum confidence level of all the certificates and servers on which it depends. Our experiments are ongoing and we believe that they will contribute significantly to the usability of our tools, but it is too early to report any conclusions in this paper.

6 Implementation Overview

There is no room here to provide detailed insights into the implementation of our prototype. In this section, however, we provide brief sketches for the most important abstractions, hoping that this will help to clarify some of the ideas presented previously. In particular, we discuss the representation of compound documents, registry objects, servers, and certificates, which are described by Java classes called `Doc`, `Registry`, `Server` and `Cert`, respectively. The code fragments below should not, however, be read as executable code; they have been been edited for this presentation to reduce clutter (for example, eliminating modifiers like `public` or `abstract`) and to elide less important implementation details.

We start with the `Doc` class, which provides methods to construct an `empty` document and to `load` or `save` a document in a specified file. It also provides methods for adding, removing, and retrieving the certificates in a document. (Note that certificates are referred to here by the `name` of the corresponding folder in the compound document.)

```
class Doc {
    static Doc empty();
    static Doc load(File file);
    boolean    save(File file);
    boolean    add(String name, Cert cert);
    void       remove(Cert cert);
    Cert       getCert(String name);
    Cert[]     getCerts();
    ...
}
```

As described in Sec.2, each different certificate type is identified by a globally unique identifier. The server for a particular `guid` can be obtained by consulting an appropriate `Registry`. Each registry can also be queried for the set of all servers that it supports and can be updated by installing new servers (or uninstalling old ones).

```
class Registry {
    Server   getServer(GUID guid);
    Server[] getServers();
    void     install(GUID guid, Server server);
    void     uninstall(Server server);
    ...
}
```

If a user tries to access a certificate on a machine where the corresponding server has not been installed, then a call to `getServer(guid)` will return `null`. Use of such certificates will be limited: some details can be extracted from its descriptor, but operations that are specific to particular certificate types—such as editing or validation—will not be possible.

Each `Server` includes attributes that specify its `guid` and a text string that can be used to describe the server in interactions with users. Once an appropriate server has been identified, the `load` method can be used to obtain the certificate corresponding to a particular `folder` in a source document. (The matching `save` functionality is located in the `Cert` class.) We can also use a server's `newCert` method to insert a new certificate of the appropriate type into a particular `host` document. In responding to this method, the server may query the user for any additional information that it needs. The server may also choose to decline the request, in which case it will return a `null` value.

```
abstract class Server {
    String getDescription();
    GUID   getGUID();
    Cert   load(Folder folder);
    Cert   newCert(Doc host);
    ...
}
```

Our prototype includes several different implementations of the basic `Server` interface, most of which correspond to the server types described in Sec.4.

Individual certificates are represented by `Cert` objects. Each certificate includes attributes that specify its server and sequent. In addition, each certificate may be associated with a particular `host` document; this association can be set or broken using the `attach` or `detach` methods, respectively.

```
abstract class Cert {
    Server  getServer();
    Sequent getSequent();
    String  getDescription();

    Doc     getHost();
    boolean attach(Doc doc);
    boolean detach(Doc doc);

    boolean save(Folder folder);

    boolean isValid();
    boolean validate();
    void    invalidate();
    boolean edit();
    ...
}
```

The last four methods support validation and editing, as described in Sec.3.2. The `isValid` method returns the value of the flag indicating whether the certificate is known to be valid, but makes no attempt to validate a certificate whose status is unknown. The latter task must, instead, be handled separately by the `validate` method. It is also possible to `invalidate` a certificate (and all of its clients) at any time, which sets the flag for each certificate to false. This will typically be used when the certificate (or something on which it depends) has been modified in some way that requires the user to recheck its validity. Finally, the `edit` method can be used to open an appropriate editor for the certificate.

7 Summary

Many tools have been developed to help programmers produce evidence that the software they are developing is correct. In this paper, we have described a new kind of tool that will help users to manage and exploit that evidence in the context of an evolving project. An initial prototype has been constructed to validate the basic design and to provide a starting point for experimentation. Areas where further research will be particularly valuable are in the exploration of techniques for building external servers, and in the development of mechanisms that will help users to organize and understand collections of evidence more effectively.

Acknowledgments

The work described in this paper was carried out in the context of the Programatica project at OGI. It has benefited and been shaped by input from several members of the Programatica team; from other members of PacSoft; and from other colleagues in the Department of Computer Science.

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. Automated Reasoning Group, University of Cambridge Computer Laboratory. The HOL98 theorem prover. `http://www.cl.cam.ac.uk/Research/HVG/HOL/`.
3. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
4. Abhay Bhorkar. A run-time assertion checker for java using JML. Technical Report TR #00-08, Department of Computer Science, Iowa State University, May 2000.
5. Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4), July 1999.
6. James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.
7. Michael D. Ernst. *Dynamically Detecting Likely Program Invariants*. PhD thesis, University of Washington, Department of Computer Science and Engineering, August 2000.
8. S.I. Feldman. Make-A program for maintaining computer programs. *Software—Practice and Experience*, 9(4), 1979.
9. Martin Fowler et al. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1999.
10. Michael J.C. Gordon. Reachability programming in HOL98 using BDDs. In *Theorem Proving in Higher Order Logics (TPHOLs)*, August 2000.
11. Thomas Hallgren et al. The Alfa proof editor. `http://www.cs.chalmers.se/~hallgren/Alfa/`.
12. Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, October 2000.
13. K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Report Technical Note 2000-002, Compaq Systems Research Center, October 2000.
14. Simon Peyton Jones and John Hughes, editors. *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*, 1999. Available from `http://www.haskell.org/definition/`.
15. E. Poll and B.P.F. Jacobs. A logic for the Java modeling language JML. Technical Report CSI-R0018, Computing Science Department Nijmegen, November 2000.
16. J.A.G.M. van der Berg and B.P.F. Jacobs. The LOOP compiler for Java and JML. Technical Report CSI-R0019, Computing Science Department Nijmegen, December 2000.