Experience Report: Evolution of a Web-Integrated Software Development and Verification Environment

Charles T. Cook, Yu-Shan Sun, and Murali Sitaraman

Technical Report RSRG-13-05
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

August 2013

Experience Report: Evolution of a Web-Integrated Software Development and Verification Environment

Charles T. Cook, Yu-Shan Sun, Murali Sitaraman

School of Computing, Clemson University, Clemson, SC 29631, USA
Correspondence: {ctcook, yushans, murali}@clemson.edu

SUMMARY

This paper summarizes our experiences over the last four years in creating a web-integrated software development and verification environment. The environment has been used for both research experimentation and education. It has been used in undergraduate computer science courses to teach modular software development and analytical reasoning principles at multiple institutions. In the process, the environment has undergone many refinements to meet demands for improved functionality and to leverage rapidly changing underlying technology for improvements in its interface and performance. The interface of the environment is tailored to present formal specifications and alternative implementations of components, and enable correctness checking through a backend verifying compiler. This paper presents a detailed account of the development and evolution of the environment—its functionality, performance, and its interface—that we hope will serve as a model for others, especially as the benefits of online learning systems are becoming increasingly obvious.

1. INTRODUCTION

The overall objective of the web-integrated environment discussed in this paper is two-fold: To help produce high quality component-based software and to make the environment widely accessible through a routine browser interface for both research experimentation and classroom education. In particular, the environment makes it possible to specify formal contracts for components, develop implementations according to those contracts, and verify formally that the implementations satisfy those contracts in a modular fashion, minimizing the cost of integration and maximizing quality. It is easy to use for it requires no installation and no understanding of underlying details, such as file directory structures, and is an open source project, hosted on GitHub. The source code is available at [1].

The features to present a user interface of component relationships, to facilitate formal reasoning on a component by component basis, and the seamless integration as a web application distinguish the development and verification environment presented in this paper from other efforts in formal methods, such as those summarized in [2], and popular IDE's, such as Eclipse and NetBeans. Multiple institutions and hundreds of students have used it over the course of 4 years. This paper discusses in detail the evolution of this environment, both in its user interface design and in its implementation, as its features were extended from a simple static demonstration site to a sophisticated, dynamic IDE with a component finder, an integrated editor, and easy access to a verifying compiler on the backend that is suitable for team software development. It should serve as a useful model for online environment design, where ease of use, timely interaction, learning, and education are among the primary goals.

The Web IDE supports component-based software development and verification in RESOLVE, an integrated specification and programming language [3][4]. The language is object-based and imperative, similar to C++ or Java, but includes constructs for formal specification. The RESOLVE Verifying Compiler helps check that an implementation is correct with respect to its specification. It generates verification conditions (VCs) of correctness for one module at a time, relying only on interface specifications of reused components; it is thus scalable. It is also able to prove the correctness of some or all of the verification conditions automatically.

The Web IDE provides an interface for component-based software development, enables the use of the verifying compiler, contains a built in JavaScript parser that will automatically report back any keyword/syntax errors, and translates all code to Java for execution. The environment, which has been used by multiple institutions for various purposes [5][6][7], has gone through several iterations, as detailed in this paper. To facilitate component-based software development and verification with formally-specified interfaces, the IDE presents components based on their relationships rather than as files, as is commonly done in many other programming IDE's. Here, the uppermost level idea is the concept (similar to a Java Interface). A concept is an interface containing only specifications. It may have one or more realizations (implementations) or enhancements (interface extensions). Like a concept, an enhancement may have one or more implementations (realizations) and contains only specifications. A facility is a unit that makes it possible to put together reusable concepts, enhancements, and their realizations to create component-based systems.

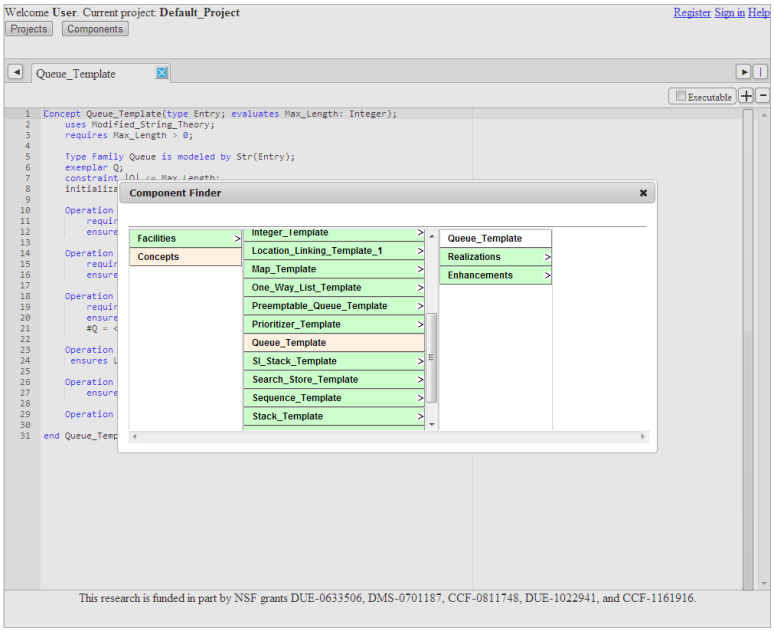

Figure 1 Component Finder Showing a Queue Specification in the Background

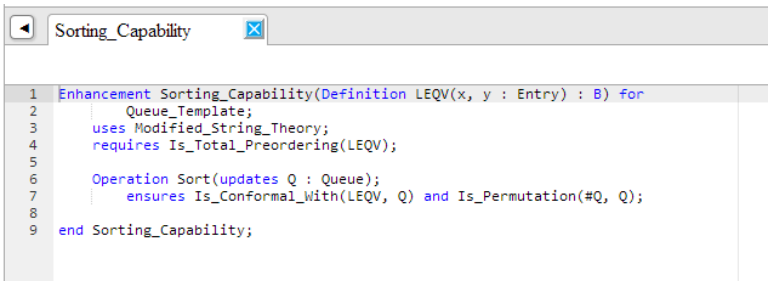

Figure 2 Specification of an Extension Operation to Sort a Queue

Figures 1 through 4 contain screenshots to provide a quick overview of the IDE interface and functionality. A detailed understanding of the technical contents of the figures is not necessary for this paper, and may be found elsewhere [8][9]. Figure 1 shows a component finder with the specification of a Queue concept in the background. Figure 2 shows the specification of a Queue sorting enhancement, whereas Figure 3 shows partial results from an attempt to verify a selection sort implementation of the sorting enhancement. In Figure 3, the blue ovals to the left of the line numbers indicate verification conditions (VCs) to be established. In general, several VCs need to be proved for implementation correctness [4] and correctness is established if and only if all VCs are proved. Figure 4 shows the result from "building" a facility that leads to translation of all corresponding files to Java and generation of a JAR file for execution.

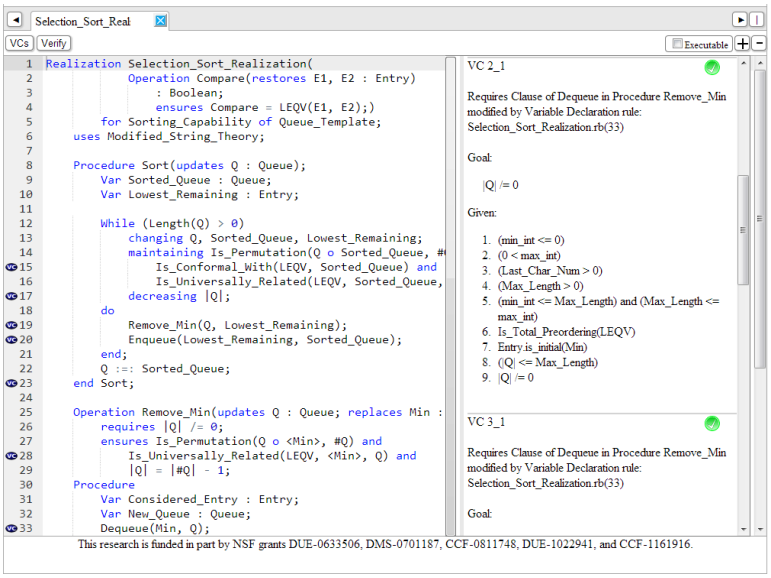

Figure 3 Partial Verifying Results of the Queue Sorting Implementation

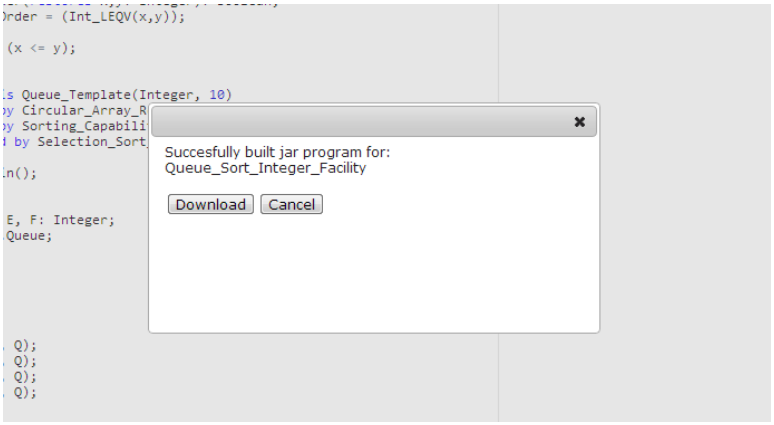

Figure 4 Building an Executable Jar from the Code

The sections below summarize the evolution of the Web IDE to meet increasing demands for functionality. They also explain how the interface of the IDE evolved with changing requirements, while leveraging advances for improved performance made possible by technological advances.

2. VERSION OF 2009

2.1. Goals

The initial idea driving the project was to create a portal for the RESOLVE Verifying Compiler (RVC) that would make use of modern web-based technologies to allow real-time editing and compilation services of components. Specific goals included the following:

- Allowing users to edit and compile components in real-time
- Providing online access to all the existing RVC functionality
- Utilizing server-side Java technologies

2.2. Design and Implementation

User Interface Design and Dynamic Content Retrieval: Modeled after the UI design for a prototype done earlier, the initial design made use of tables for basic formatting and was made up of two sections, as shown in Figure 5. The upper section contained four select boxes. The lower was made up of a text area and a column of user control buttons. To present the component relationships to our users, each of the four select boxes in the upper row was reserved for a specific component type—concepts, concept realizations, enhancements, and enhancement realizations.

The components shown in the boxes would be populated dynamically via AJAX (Asynchronous JavaScript and XML). When opening the interface users would initially see only concepts; a JavaScript function would be called on page load that sends an initial request to the server via AJAX. A Java Servlet would receive the request, open the “Concept” directory, and send a response containing a list of the subdirectory names back to the client’s browser. The response triggers a JavaScript handling function that uses the list of concepts to create an HTML option block for each of the concepts in the list. The options code would then be inserted into the concept select box for display to the user. The user would then click on a concept name,

triggering another AJAX request to the server containing the name of the clicked concept as a parameter. The servlet would use the name of the target concept to retrieve its code and create lists of any available concept realizations or enhancements based off the contents of those respective sub-directories. The response triggers a function to populate the concept realization and enhancement select boxes and insert the code in the text area below, as seen in Figure 5.

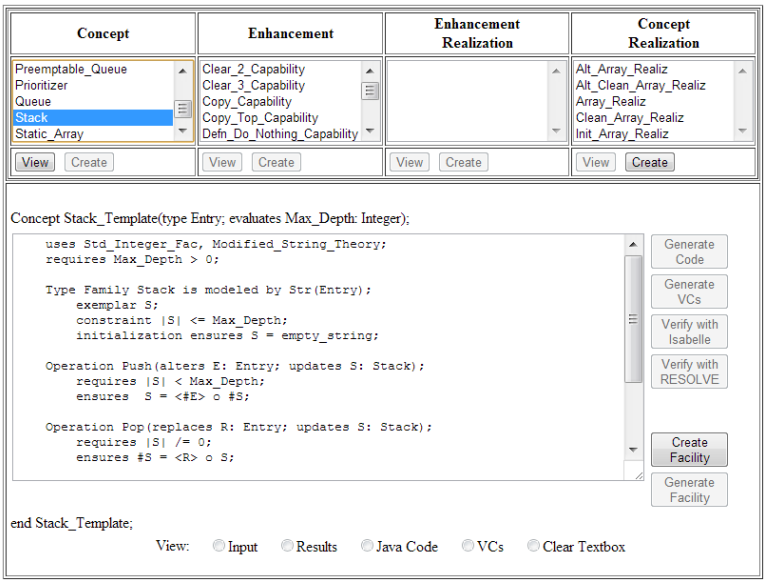

Figure 5 2009 Web IDE

Now that a component is selected and visible, the user is able to make use of RVC functionality. Based on what type of component has been selected, the appropriate RVC command buttons are activated.

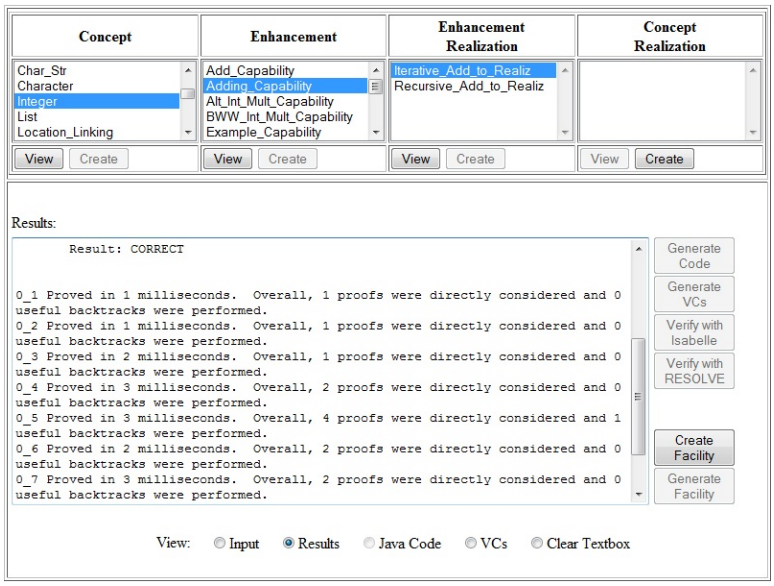

Figure 6 Sample Output from 2009 Web IDE

Server-side Implementation Details: In the initial design, we chose to use the Apache Tomcat container to serve as an intermediary between the Apache web server and the Java-based RVC. Using server-side Java would allow us to provide seamless integration with the RVC, instead of having to rely on using server-side scripts to execute the program as if it were run via the command line. This would also provide us with the ability to pass complex objects to the RVC for communication, as opposed to relying on parsing piped command line output. Finally, instantiating the RVC would provide the capability for long running tasks, such as component proving, to be run independently and asynchronously, allowing us to implement the server in a way that doesn't freeze up the user interface while waiting for the server's response.

Server-side RVC Interaction Details: When a command button is selected (as seen in Figure 6), the servlet handling the RVC interactions must take several actions before dispatching the task to the RVC. For example, the command requires the servlet to write all RVC target components temporarily within the RVC's workspace directory. To avoid naming conflicts, the backend must first generate and remember a random name for the component. Since the RVC has a strict filename enforcement system, the servlet must then replace the actual name in the source code with the random name before writing the file to disk. It would then invoke the RVC and wait for completion. Once finished, the servlet begins composing an XML response and checks for a Java file of the same (random) name in the same directory as the newly written source file. If the Java file is not found, it reports failure in the response; otherwise it reads in the Java file, encodes the file contents for URL transmission, and incudes it in the XML response. When the JavaScript handler function is triggered, it first checks whether the job is successful. If so, the results are decoded, saved, and inserted into the text area for display and the radio button indicator is activated and switched appropriately. If the command failed, error messages are displayed. The user would then be able to toggle between the input and RVC output by clicking the appropriate radio buttons. Figure 6 shows the results after navigating to a realization and clicking the "Verify with RESOLVE" button. The logic and handling for all server tasks was the same—clicking any of the activated command buttons would send the text area source to the server and display the output from the compiler in the text area, while activating the appropriate radio button for the task.

2.3. Discussion

The December 2009 version of the Web IDE provided access to the basic functionality of the RVC for the four types of components available. It supported translation to Java for all modules, however VC generation and verification were limited to enhancement realizations. It was used in the spring of 2010 software engineering courses at Clemson. Instructors were able to use the hierarchical nature of the component select boxes to emphasize the relationships between the components and to utilize the ability to make changes to components, invoke the server-side compiler, and see the results in real time. While this version of the interface met all the development goals, user feedback and post-release analysis allowed us to create a list of design shortfalls in functionality, performance, and user interface design.

A key missing feature was the ability for users to create new artifacts. There was also a significant performance issue. Each time a user would click a component option, a request was dispatched to the server to retrieve both the contents of the selected file and any associated files. This task required several disk accesses for each component the user clicked, resulting in unnecessary server overhead in terms of both processing time and disk I/O. Also, with the user

interface design, screen space was not used effectively, because more often than not, the concept realization and enhancement realization select boxes were either empty or contained only a couple of options.

3. VERSION OF 2010

3.1.Goals

The version of 2010 was designed to address issues raised above and provide a more modern and intuitive user interface, with the following goals:

- Visually overhaul the user interface
- Redesign the backend to be more flexible and less reliant on disk I/O to improve performance
- Provide increased functionality by supporting user-created components

3.2. Design and Implementation

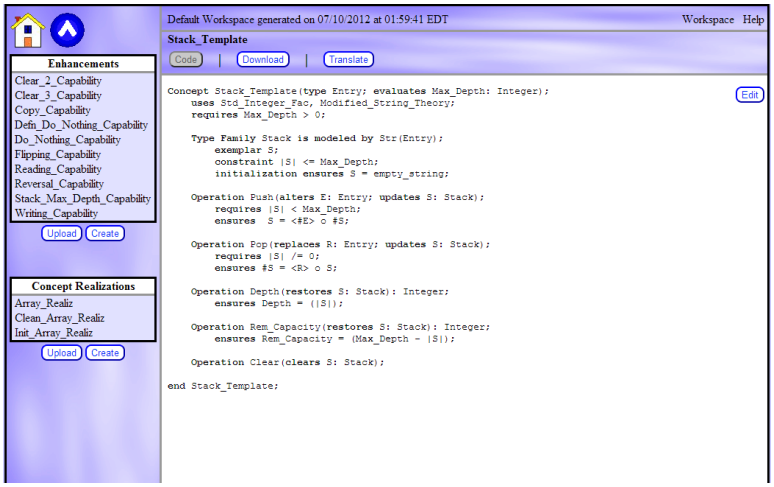

Figure 7 Redesigned 2010 Web IDE Interface

User Interface Redesign Details: The overall goal of the revised UI redesign was to take better advantage of available screen space and to use CSS (cascading style sheets) to format the page instead of using HTML tables. This design eliminated the component select boxes and moved the list of available components to the left side of the interface, allowing the code to take up almost the whole height of the interface. In addition, it took a more dynamic approach to displaying the components. Now users would initially only see menus of available concepts and facilities (component-based systems built from reusable components) along the left side of the page. When a concept name, for example, was selected, a function was called that removed the concept and facility lists from view and repopulated the container with lists of available concept realizations and enhancements. The function would also display the concept to the right of the menus in the code editor pane. To be more user friendly, code was now preprocessed by wrapping all RESOLVE keywords with HTML b tags and replacing formatting characters, such as spaces and newlines, with the equivalent HTML character codes and inserted into the pane as

seen in Figure 7. An edit button was also added at the far right that would replace the rich text code with an editable text area containing plain text code.

Server-side Additions and Modifications for Functionality and Performance Improvements: In addition to the client side UI changes, the server implementation was also revamped. The "Concept" directory hierarchy was abandoned in favor of an XML file containing everything needed to represent the relationships between the components in the RVC workspace. A web service was created to generate this file in advance by scanning the components within the workspace, analyzing the text of the files within each to determine their relationships, and outputting the generated hierarchy (including the actual code for each component) as an XML document, a snippet of which is shown in Figure 8.

```
<config>
    <concepts>
        <concept>
            <name>Queue_Template</name>
            <body>RESOLVE code ...</body>
            <enhancements></enhancements>
            <realizations></realizations>
        </concept>
        <concept>
            <name>Stack_Template</name><body>RESOLVE code ...</body>
            <enhancements>
                <enhancement><name>Clear_2_Capability</name><body>RESOLVE code ...</body>
                    <realizations>
                        <realization><name>Clear_2_Realiz</name><body>RESOLVE code ...</body></realization>
                        ...
                    </realizations>
                </enhancement>
                <enhancement><name>Copy_Capability</name><body>RESOLVE code ...</body>
                    <realizations></realizations>
                </enhancement>
            </enhancements>
            <realizations>
                <realization><name>Array_Realiz</name><body>RESOLVE code ...</body></realization>
                ...
            </realizations>
        </concept>
        ...
    </concepts>
</config>
```

Figure 8 A XML Snippet for a User Workspace

When the Web IDE was opened, the XML file was sent in a compressed format directly to the client's browser. The XML was then parsed to create a JavaScript array of components that would ultimately be used to populate the component menus as needed. This new technique had several additional advantages. The most apparent was performance improvement. Because the user interface received everything it needed upon initialization, it no longer needed to send I/O intensive requests to the server each time a user selected a component. Instead, the component would be retrieved from the array and the information was used to repopulate the component menus and code block. Maintaining the entire component hierarchy locally also enabled new functionality. Users could create and save new components temporarily, where they would still be accessible after selecting another component. The only other changes needed were to maintain an additional array keeping track of new components and to modify the RVC job handler and AJAX functions to send and process them on the server, when a user initiates an RVC job.

Servlet Sessions for Persistence: Support for servlet sessions was added to the server implementation to facilitate persistence. Using sessions allowed us to provide a level of UI persistence not previously possible. In this design, prior to initializing the UI, the server first checks a server-side session variable for the XML representation. If not found, the server reads the contents of the XML file into the session variable String. Only then does the server then send the representation string to the user interface. This reduces the amount of disk I/O necessary; if the user simply refreshes the page in their browser there is no need to access the physical disk. To make the most of this session variable, the user interface was also given the capability to

recreate the XML string from the stored components (including any user-created components) and transmit it to the server. The server would then update the session variable with the updated string. This allowed the system to maintain the state of the user interface across browser refreshes. The next step was to provide UI persistence across browser and computer restarts. To achieve this level of persistence, a handling servlet was added that allowed users to download the XML representation from session memory to their local file system and to import and reload a previously downloaded XML file.

Servlet Response-Handling Improvements: In addition to updating component navigation and display, RVC results from the server were also improved. Results from a selected "VCs" or "Verify" button were now pre-processed prior to display. When the VC job response was received, pattern matching was used to break the response into an array of VCs, each containing the VC, the source code line number information, and the VC details. This array was then used to generate the output. Initially, as seen in Figure 9, while only VC index and line information is visible for each VC, a custom toggle button makes it possible to view or hide the VC details on demand. Also, the verification conditions were generated and displayed to the user, even as the automated verification process to prove them was underway, which could take a significant amount of time depending on the complexity of the VCs. Once proving was complete, the RVC output is again analyzed and the results are overlaid as an image onto the appropriate VC.

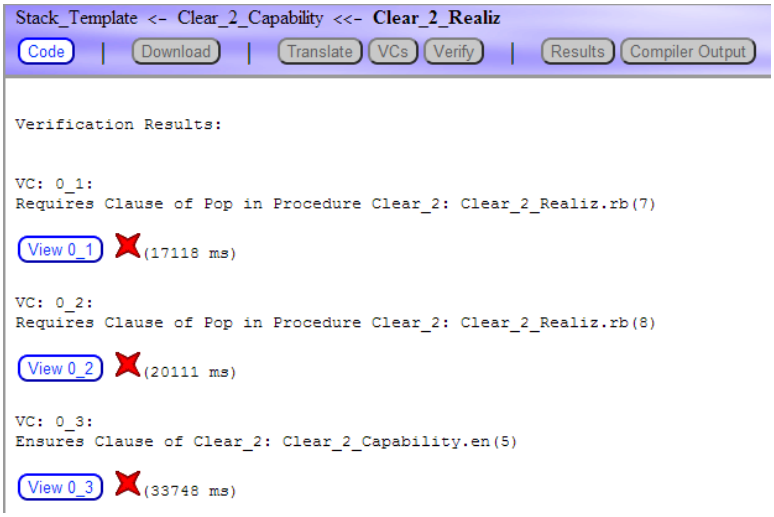

Figure 9 Improved Display of Verification Results

In this version, the ability to create an executable JAR (Java archive) for an entire component-based system was also added to the RVC and the Web IDE. The IDE included a "Build" button that would produce a JAR file, which could then be downloaded to the local file system for execution.

3.3. Discussion

This version was deployed for use in software engineering courses in the fall 2010 semester at Clemson. The keyword highlighting and more appealing display of verification conditions made this version much more useful as a demonstration tool. In addition, the ability to create and use new components allowed instructors to use the interface for a project that required

students to create a RESOLVE program using both existing components and new ones created by the students as part of the assignment. Treating the interface as a development environment for the project allowed students to experience the entire process—from implementing reusable modules based on given specifications to executing the finished product. For more details on the software engineering assignments at Clemson and elsewhere, see [9]. Even though feedback was positive, there were more areas for improvement in terms of functionality, performance, and interface design. These improvements include real-time error highlighting, minimizing disk accesses by taking advantage of browser memory, and project management features.

4. VERSION OF 2011

4.1. Goals

In addition to modernizing the visual aspects of the user interface, this version would be designed to take advantage of advanced capabilities built into many modern browsers to provide an enriched user experience:

- Modernize the user interface
- Leverage advanced browser capabilities to improve performance
- Make use of 3rd party libraries to provide additional functionality

4.2. Design and Implementation

User Interface Redesign with Reuse: One of the primary goals of the visual overhaul was to make the interface look and behave more like a modern development environment, allowing users to edit and view multiple components more easily. This redesign mimics the basic behavior of most modern web browsers by using tabs. There would be five horizontal tabs; each of which would be reserved for a specific component type (facility, concept, concept realization, enhancement, and enhancement realization). To increase the usable space, the component menu that was always visible was removed in favor of modern dropdown menus, accessed via dropdown icons at the top of each tab. When a user selected a tab, the code would be displayed in the tab body.

The more prominent change had to do with how components are displayed and edited; the ACE editor [10], a fully JavaScript/CSS-based open source rich-text code editor, is now embedded within the UI (Figure 10). The editor supports keyword highlighting for a large variety of programming languages and real time syntax error checking for JavaScript. We extended the native capabilities of the editor by creating a custom add-in module providing the rules for RESOLVE-style keywords and constructs, allowing keywords and other basic structures, such as single and multi-line comments, to be color-coded appropriately. Additionally, the RVC's ANTLR-based grammar was leveraged for use in the Web IDE. With little modification to the grammar's ANTLR source files they could be used to generate a JavaScript parser. This parser was then integrated into the user interface to provide real-time syntax checking for currently open components.

Since this version of the user interface was being rebuilt from the ground up, we decided to take advantage of jQuery [11], a popular and commonly used external library. This allowed us to spend more time focusing on properly implementing the core functionality of the user interface and less time with common tasks such as event handling and dynamic manipulation of

DOM elements. It also made sense to use the jQuery UI plugin to create the tabbed user experience of a modern UI. The UI plugin provides all the functions necessary to visually create and manipulate the component tabs. In addition to the component-level tabs, component source is displayed in a sub-tab and the RVC results are displayed in additional tabs, dynamically created on demand. This allows users to quickly and intuitively switch back and forth between the source and results.

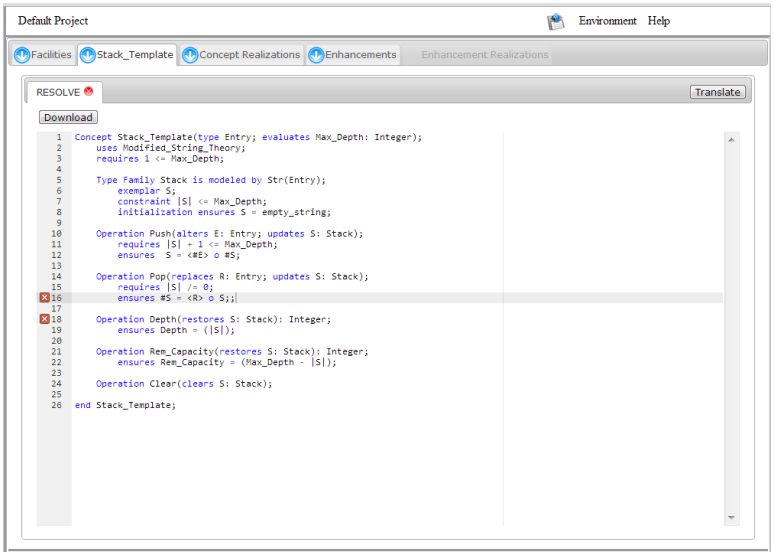

Figure 10 Integration of a Rich Text Editor with Real-Time Syntax Checker

Leveraging Modern Browser Features to Improve Performance: One of the new browser capabilities leveraged in this redesign was the File API, which allowed the client-side code to interact directly with the local file system. Drag-and-drop file loading was also included, allowing a user to load an exported XML representation into the environment by dropping it onto a special icon in the user interface. Instead of server-side session storage, this redesign used browser local storage to store and update the XML representation. Storing this data locally reduces the communication to disk I/O required on the server. The only time the XML file needs to be read in from disk is if the user has never used the IDE before or is resetting their environment to the default state. In addition to providing persistence across browser refreshes, this technique provides persistence across browser and computer restarts without requiring users to download and re-import the XML representation.

Enhanced Features: In addition to the integration of a rich text editor, several new features were also added to this version. One improvement was a tighter visual integration between the generated VC and the line of source code that prompted its generation. For each statement that led to the generation of a VC, a custom VC icon was displayed on the line that spurred the generation of the VC. When a user hovered over the icon, a brief description of the VC would be displayed, as shown in Figure 11.

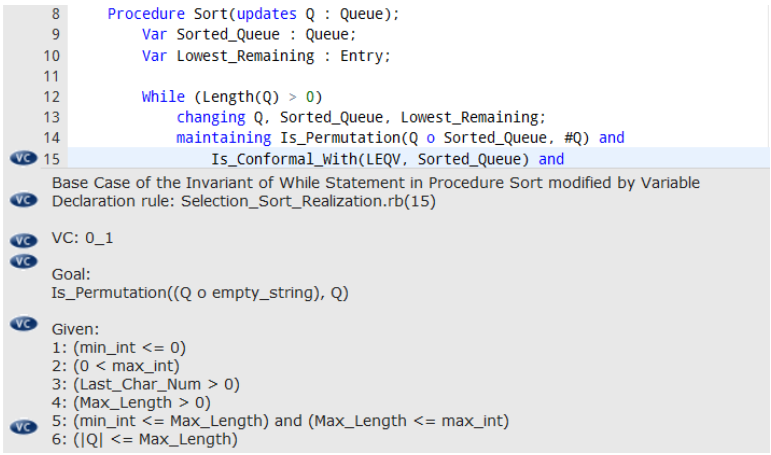

Figure 11 Showing Verification Condition in Context

Another new feature was the ability to select from and display multiple projects based on different workspaces (depending on background and knowledge of the audiences). Adding this support allowed us to provide libraries containing a variety of different components or components with either different specifications or implementations with the same names. This required changes to both the server side implementation and the front end user interface.

A login system was also created for the Web IDE to provide support for public and private projects. Java Servlets were added for rendering web pages for and handling login, logout, user registration, and password recovery. Additionally a servlet now generated the basic HTML code for the user interface. Server-side control made it possible to customize what UI elements are shown based on user status. To support the user login system, a simple MSQYL database was added on the server. The database was primarily responsible for maintaining registered user information and as a data store for user actions.

4.3. Discussion

This redesign was completed and used in software engineering courses at Clemson and several partner universities in the spring of 2011 for expanded software engineering assignments. While all of the students were ultimately able to complete the assignment, there was feedback for improvement. One confusing issue for students was that they had assumed that because they were logging into the IDE, their components would be available from any machine they logged in from. However, this was not the way the system was implemented; user components only existed in the local browser storage and the login system had been included only to allow restricted access to projects. While this was a non-issue for students who were working exclusively from their personal laptops or saved their work on files, it was a problem for those who were using lab computers. This issue indicated that user-created components should be stored in such a way that they can be accessed from any computer.

Another point of feedback was the inefficiency involved in switching to view one component from another that required a traversal (through many clicks) to the parent concept. Although this was how component navigation was intended and was natural for those familiar with RESOLVE's component hierarchy, it was ultimately an interface design flaw.

5. VERSION OF 2012

5.1. Goals

After several semesters of use by researchers and students, the current version—as depicted in the introduction section, has been developed. In addition to expanding the functionality, this redesign has adopted the Model View Controller (MVC) design pattern and made use of MVC frameworks to facilitate further evolution with ease. Prior to beginning development, the following goals were identified:

- Take advantage of MVC frameworks to reduce development and evolution time
- Make use of HTML Web Sockets to enable real-time, two-way client/server messaging
- Provide server-side storage of user-created components

5.2. Design and Implementation

Client-side Framework Redesign to Ease Further Development and Evolution: To assist in the redesign of the user interface, we chose to make use of Backbone.js [12], a JavaScript library designed to provide client-side MVC-like functionality for web applications. In this version, the information for a component would be stored as a relational model object, an extension of the Backbone.js general model provided by the Backbone-Relational extension library. The relational model adds support for one-to-many mapping between models and is thus ideally suited for storing a component and maintaining its context within the component hierarchy. Now, storing the entire component hierarchy is as simple as creating a Backbone.js collection of relational models of concepts, each of which contains collections for both enhancements and realizations. This also provides the ability to bootstrap more easily at the initialization time. The collection constructor allows a JSON (JavaScript Object Notation) object to be passed in as a parameter. To facilitate bootstrapping the component collection, the XML representation generation service has been adapted to generate a JSON representation suitable for generating the collection. In addition to the benefit received from easily bootstrapping components, using JSON-encoded transmissions (with a smaller intrinsic file size than XML) serves to shorten the initialization time for the user interface.

Views from Backbone.js are also leveraged in this version of the IDE. The view is primarily responsible for rendering HTML code for a collection or model. A view automatically re-renders based on a change to the model to which it is bound. Thus views have been utilized wherever possible to simplify development and evolution.

In an effort to expand the capabilities and usability of the interface, fixed component type tabs have been eliminated in favor of a more free-form editing space. Instead of dedicated tabs with dropdown menus, a finder has been devised as shown in Figure 1. The Finder allows users to browse through all of the components with having to first select and open a parent concept. The HTML code for the lists in the finder is generated exclusively using Backbone.js views. Views are created for and assigned to the master component collection that rendered the highest level list (concepts, facilities, and theories that are used in mathematical specification of concepts) and for the model representing the components. Creating the default content for the finder is as simple as calling the render function for the collection view. In addition to a function for rendering code, the view contained built-in handling of typical user interactions. As a link in the Finder was clicked, concepts, for example, a new empty HTML list is generated; the

collection view recognized the click and automatically called the render function for each view (concept in the example). The model view would query the model for its name and generate and return the HTML list item code to the collection view, which inserted the completed list into the Finder dialog. Besides the Finder dialog's simplification of component browsing, it provides another advantage to users—once the menus are populated they are maintained even if the Finder is closed. The user can pick up browsing where they left off.

Server-side Framework and Web Sockets: To realize our other goals, we evaluated the current state of real-time server communications using Web Sockets. While the major Java servlet containers all implement web sockets, the APIs are sufficiently different making it impossible to tie applications to a particular server. Unfortunately, there are no third party frameworks that provide an abstraction that is simple enough to be easily dropped into an application. For these reasons, this redesign uses a server-side MVC framework, Play 1.2 [13]. In addition to its MVC capabilities, Play has several other benefits. It is also a standalone web server application with built-in support for advanced web technologies, including web sockets. The built-in application server could be run in two modes—development or production. Using development mode to write our server-side Java code leverages the framework's just-in-time Java compiling capabilities. Instead of recompiling all of the Java code and restarting the Tomcat server, simply refreshing the browser page signals the framework to recompile and reload only classes whose source had been modified.

Since web sockets are needed to provide real-time updates for component verification on a VC by VC basis, for simplicity and code reusability web sockets are used for all RVC jobs. While some of the servlet implementation code from the previous version could be reused in Play Controllers, a few new additions were necessary. The first was a web socket management controller that would handle opening and closing the connection, sending and receiving socket messages, and routing and dispatching RVC jobs. The controller would pass a handle to the output socket directly to the RVC (In the case of a verification job) which would be used to transmit the result in real-time directly to the browser. The browser would receive the message, read the VC id, and output results, as seen in Figure 3.

Login System Improvements: The Play framework is also beneficial because it provides modules for user registration, login, logout, and password recovery. It includes an API for working with the current login state within interface code. With a fully developed and integrated login system using Play, a decision was made to allow only registered users to be able to create and save new components. With this decision, all components created by users would be stored in the application's server-side database, as opposed to client-side browser storage. Once saved, the components are always available to the users who created them, regardless of what computers or browsers they may be using. Since all user-created components are stored on the server and already available to the RVC, there is less data transmitted for each action, making typical user operations much more efficient.

5.3. Discussion

The most recent version has been use in classrooms since fall 2012. While we continue to receive feedback for improvements, the current system is stable and being reliably used. Surveys continue to be administered across several semesters to receive feedback on the IDE's usability and suggestions for improvement. Software engineering students show a high level of agreement (with a score of about 4 on a likert scale of 5) with statements, such as, "The web interface has

helped reinforce the principles discussed in class," and "The web interface has helped me understand the relationship between mathematical reasoning and software development", demonstrating the benefits of the IDE [5].

6. RELATED WORK AND CONCLUSIONS

There are several, ongoing efforts to use the web as a software development platform and numerous efforts with the goal of automating software verification based on a variety of specification and programming languages [2]. However, there are few efforts that combine both. In detailing our experience over the evolution of such a combination, this paper provides a roadmap of issues and solutions for future developers.

Two systems that facilitate web-based software development and verification are Dafny and VeriFast. Dafny is a full strength programming language developed at Microsoft [14]. It uses Boogie, an intermediary language, to generate verification conditions and passes them to an automated prover, Z3 [15]. VeriFast is a verification system that allows the user to add annotations within Java and C programs and uses an interactive prover that uses Z3 under the hood to check for verification errors [16]. Both Dafny and VeriFast have online user interfaces with examples that users can take advantage of without having to install anything on their machines. Users can create new programs and obtain verification results within the browser. Another related system is PEST [17]. Though it does not include a web interface, PEST provides a basic programming language to teach software verification in introductory courses. There is an Eclipse plugin for PEST to ease its use.

Several web-based efforts focus on software development with popular languages, such as C++, Java, JavaScript, and Python, though few have been used in classrooms or for research experimentation on a regular basis. Cloud9 IDE provides real-time code analysis and debugging tools for JavaScript-based web application development [18]. The IDE supports team development through simultaneous file editing and a built-in instant messaging system. ShiftEdit includes real-time syntax error checking [19]. It does not host any user developed files on its servers, but provides support for uploading to FTP/SFTP servers and online services like Dropbox, Google Drive, and Amazon S3. It also allows optional collaboration via sharing and revision history. Compilr is geared more toward compiled languages, such as C#, Visual Basic, Java, and C++, though there is some support for interpreted languages PHP and Ruby [20]. Coderun requires no registration, but to save projects and files to the server, users must register and log in. It provides syntax coloring for several languages, but only variants of C# have code completion, server side debugging, and compilation [21]. Ideone is a service that has a large catalog of over 40 languages that can be developed through their web-based IDE and executed on the server side through a submission-based user interface [22]. Registered users have access to the Ideone API, allowing them access to Ideone's submission-based backend within their own applications.

This paper has summarized the experiences in building and using a web-integrated system for development and verification of component-based systems. It has detailed the evolution of the environment over four years, as the system changed from a simple demonstration vehicle to a full-fledged IDE. The need to have a reliable working system available for use under stringent educational calendar deadlines every year, even as the requirements changed, has led to numerous challenges as detailed in this paper. The paper discusses a sequence of refinements in detail with the aim of serving as a model for evolving web-based systems. The presentation explains the changes in demands on functionality and the

impact of such changes on interface design and performance, for each refinement. In the process, it describes how rapid web-technological advances have been leveraged to address the demands.

ACKNOWLEDGEMENTS

We thank past and present members of our research groups at Clemson and Ohio State for their contributions to the ideas contained in this paper. This research has been funded in part by the NSF grants CCF-0811748, CCF-1161916, DUE-1022191, and DUE-1022941.

REFERENCES

[1] RESOLVE Web IDE hosted by GitHub. [online] Available at: https://github.com/ClemsonRSRG/RESOLVEWebIDE

[2] Klebanov, V., Muller, P., Shankar, N., Leavens, G. T., Wustholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K. R. M., Monahan, R., Piessens, F., Polikaropva, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., and Weiss, B., "The 1st Verified Software Competition: Experience Report," *FM 2011: Formal Methods (Proceedings 17th International Symposium on Formal Methods)*, Springer LNCS 6664, June 2011, 154-164.

[3] Sitaraman, M., and Weide, B. W., "Special Feature: Component-Based Software Using RESOLVE," *ACM SIGSOFT Software Engineering Notes 19*, 1994, 21-67.

[4] Sitaraman, M., Adcock, B., Avigad, J., Bronish, D., Bucci, P., Frazier, D., Friedman, H. M., Harton, H., Heym, W., Kirschenbaum, J., Krone, J., Smith, H., and Weide, B. W., "Building a Push button RESOLVE Verifier: Progress and Challenges," In *Formal Aspects of Computing*, Springer, 2011, 607-626.

[5] Cook, C. T., *A Web-Integrated Environment for Component-Based Reasoning*, M. S. Thesis, Clemson, September 2011.

[6] Drachova-Strang, S., *Teaching and Assessment of Mathematical Principles for Software Correctness Using a Reasoning Concept Inventory*, Ph. D. Dissertation, Clemson, 2013.

[7] Smith, H., *Engineering Specifications and Mathematics for Verified Software*, Ph. D. Dissertation, Clemson, 2013.

[8] Cook, C. T., Harton, H. K., Smith, H., and Sitaraman, M., "Specification engineering and modular verification using a web-integrated verifying compiler," *ICSE*, Zurich, 2012, 1379-1382.

[9] Cook, C. T., Drachova-Strang, S., Sun, Y., Sitaraman, M., Carver, J. C., and Hollingsworth, J. E., "Specification and Reasoning in SE Projects Using a Web IDE", In CSSE&T 2013, 229-238.

[10] ACE Editor, ACE Editor: The High Performance Code Editor for the Web. [online] Available at: http://ace.ajax.org/#nav=about

[11] jQuery User Interface (UI). [online] Available at: http://jqueryui.com

[12] Backbone.js. [online] Available at: http://backbonejs.org/

[13] Play Framework – The High Velocity Web Framework for Java and Scala. [online] Available at: http://www.playframework.com

[14] Leino, Rustan K. M. "Dafny: an automatic program verifier for functional correctness". In Proc. *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. Springer-Verlag. 2010. p. 348-370.

[15] Barnet M., e. al. “Boogie: A Modular Reusable Verifier for Object Oriented Programs”. In *Formal Methods for Components and Objects*. LNCS 4709. 2006. P. 364-387.
[16] Jacobs B., et al. “A Quick Tour of the VeriFast Program Verifier”. In Proc. *APLAS 2010*, tool paper track. LNCS 6461. 2010. P. 304-311.
[17] de Caso, G., Garbervetsky, D., and Gorín, D., “Integrated program verification tools in education”. In *Software—Practice and Experience*. John Wiley & Sons, Ltd. 2013, 43(4): 403-418.
[18] Cloud9 IDE, Inc. 2011. Cloud9 – Your code anywhere, anytime. [online] Available at: http://cloud9ide.com
[19] ShiftEdit. ShiftEdit – Online IDE. [online] Available at: http://shiftedit.net
[20] Compilr. 2011. Online C#, PHP, C, C++, Ruby, VB, Java IDE & Compiler. [online] Available at: http://compilr.com
[21] Coderun Studio. 2009. Web Development and Deployment Tools: Coderun. [online] Available at: http://www.coderun.com
[22] Sphere Research Labs. 2011. Idone.com | Online IDE & Debugging Tool. [online] Available at: http://ideone.com