Flexible Interface for Automated Proofs

Nate Soule
Boston University
Computer Science Department
Boston, MA 02215, USA
nsoule@bu.edu

ABSTRACT
The aartifact system [1] is an automated assistant for formal reasoning. It is currently architected using a client-server model. In order to better support in-class use, and enable greater scalability a JavaScript implementation of key functional areas of the system is desired. The automated proof landscape has to date concentrated little effort towards creating suitably usable interfaces. Towards this end, usability was one of the core goals of aartifact. In the process of moving towards a richer client side system, usability is again a highlighted design goal. This paper details the construction of a browser based JavaScript implementation of a subset of aartifact’s parsing and validation engine, along with extensions to further the goals of scalability and usability.

I. INTRODUCTION
The automated assistant for formal reasoning system aartifact [1] is a tool for exposing powerful logical and mathematical reasoning in an intuitive and expressive interface. The current state of largely inaccessible formal verification tools led to an emphasis on natural representations and user interactions during the development of aartifact. Designed and architected as a lightweight web front end, the core processing of aartifact occurs server side. In order to support scalability, among other benefits, a transition from a mostly server resident system, to a predominantly client based tool was desired. This work details the implementation of a browser resident JavaScript parsing and validation engine to be potentially used as the basis for such functionality in aartifact, as well as numerous user interface driven designs for potential inclusion.

I.1 The System
A web based JavaScript implementation was created to provide two core features of aartifact: parsing, and validation. The new system parses user input and validates the resulting data structure using browser resources. No server component is included. The user input takes the form of propositional logic, extended with the keywords “Assume” and “Assert” for providing facts to be assumed, and statements to be validated, respectively. This input language is a subset of that currently allowed for in aartifact. The code of the new system was written in a modular way, however, so as to allow for implementations of the implication and execution engines to act as plug-ins and thus allows the system to be easily extended to more powerful logics.

I.2 UI Capabilities
The new system incorporates several user interface capabilities intended to both ease the usage of such a tool, and potentially to allow for greater scalability by allowing for partial validation checks. These features are briefly outlined below:

Partial Proofs. The concept of a partial proof was introduced into the system. A partial proof can represent any sub-section of the overall proof under construction. The partial proof is represented by a widget in the user interface that can be created and placed anywhere on the “canvas” of the tool. This is intended to more closely model the natural proof writing methodologies common in hand written proofs. When working with paper and pencil a proof writer is not limited to working left to right, top to bottom on the page. They may, for example, create several lemmas on various areas of the page, and then return to the very bottom to combine them into the higher level proof. Partial proof widgets provide this ability in a web based proof environment. Users may create partial proof widgets in locations of their choosing, and unlike in the paper-and-pencil form of proofs they may then choose to organize the entire layout by clicking an "Organize Proof” button at the top of the screen. This function currently provides a simple organizational layout by sorting the partial proof boxes
Local and Global Contexts. The introduction of partial proof widgets as the sole entry point for user input opened up the possibility of providing simplification and potential performance benefits by considering each partial proof as a separate context. Here each partial proof has a local context where it’s assumptions are stored, and where it’s conclusions are ultimately added. When creating and working with a partial proof two drop down boxes are present, Import, and Export. These refer to the particular proof box’s desire to import assumptions/conclusions only from it’s local context, or from a larger global context, and similarly to export it’s assumptions/conclusions only to it’s local context, or to the global context. In this way proof boxes representing a portion of an overall proof may be made to only consider the assumptions present within it when determining validity, or to import the assumptions and conclusions from other proof boxes on the page. By restricting the imports to the local context, and thus reducing the number of statements to consider during the validation phase, the processing can be expedited. Likewise, if the conclusions of this particular proof box are only used within this box (as it represents the final result of the proof, or a corollary not needed elsewhere), it can chose to only export them locally, and thus not affect the validation processing time of the rest of the proof.

Provenance. In order to support the concept of local and global contexts it was helpful to add provenance data to the system. In this context provenance refers to which partial proof(s) contributed to a particular statement. In the case of an assumption statement, the element clearly was contributed by the proof box that contains it. In cases of statements that are implied by other statements, however, it may be the case that \( P \) from box 1, and \( Q \) from box true implied \( P \land Q \), and thus both boxes 1 and 2 would appear in the lineage of the statement \( P \land Q \). Provenance data is helpful in determining what statements need to be removed from the global context when a local context changes, allowing for incremental rebuilds of boxes that relied on the global context. In addition, however, it may be useful debugging information to a user, if they can for example see clearly why \( P \land Q \) is determined to be valid by the system.

Inline Feedback. To maximize the available screen real estate inline feedback is provided. When an assertion is deemed invalid it is colored in red. By hovering the mouse over the failed assertion the user can see a popup box stating why the assertion failed (which parts failed to be proved valid, and their constituent failures if the failed clause is composite). To see all validation errors across the entire proof, the user can click on a "Show errors" button at the top of the screen. This button is only present when the current status of the overall proof (as represented by the status text in the toolbar at the top of the screen) is set to Invalid. Hovering over green, successfully validated statements, does not currently provide further feedback, however this would be a strong candidate location to display provenance information in future enhancements.

Inline Text Modifications. Beyond the textual modifications occurring for inline feedback, as discussed above, other modifications may be made for the purposes of more comprehensible reading of the input statements. When a user stops actively editing a particular proof box (by editing another, or simply clicking outside of the active box), the logical connectives: \( \land \), \( \lor \), \( \neg \), \( \rightarrow \) respectively. This allows for a clean visual representation of the statements, while still accepting standard keyboard characters as input from the user. The graphical representations will revert to the textual when the proof box becomes active again for editing.

II. USAGE

Figure 1 displays a screen shot of the tool in use. The screen is organized around a "canvas" the outermost container of proof widgets. Within the canvas multiple partial proofs are displayed. At any given time a single partial proof can be actively edited. In the screenshot, the topmost proof box is active, as denoted by the draggable window frame surrounding it, the toolbar, and the raw text of the proof. In comparison, the proof box below is inactive and so has no visible controls, and in addition the text has been modified from the raw form to display graphical representations of the logical connectives, as well as inline feedback regarding the validity of the assertions. Here, Assert \([\Box]\) is green, as it follows logically from the stated assumptions.

Clicking the "Validate" button of a proof box, or alternatively simply clicking outside of the proof box, causes validation to occur. If the proof box currently being transitioned into inactive mode contributes to the global context then this context will be updated appropriately with any new conclusions (and removing any old that no longer hold). Subsequently any proof boxes that import from the global context will have a chance to re-validate given the new state of the system.

Example Use Case. In an example use case a student learning propositional logic wants to test out one of De Morgan's laws. They point their web browser to the correct URL and are presented with a blank canvas. In this simple case they require only a single partial proof wid-
Figure 1: A screenshot from the tool

get. They right-click to place the widget in the desired location on the canvas. As they type the text below, the widget expands/contracts appropriately to fit the entered input.

Assume \{P \land [Q]\} or [R].
Assert \{(P) or [R]\} and \{(Q) or [R]\}.
Assume \{(P) or [R]\} and \{(Q) or [R]\}.
Assert \{(P) or [Q]\} and \{(R) or [Q]\}.
Assume \{(PA) and [PB]\} and \{(QA) and [QB]\} or \{(RA) and [RB]\}.
Assert \{(PA) and [PB]\} or \{(RA) and [RB]\} and \{(QA) and [QB]\} or \{(RA) and [RB]\}.

Progressing in highest complexity to lowest, this output shows that the system was unable to prove the portion of the assertion consisting of the grouped \{Q and R\} clause. It was unable to prove that since it could not prove the ungrouped version of the same clause. The ungrouped version was unable to be proved since it could not prove Q (a necessary step in the proof of Q and R). Likewise it could not prove R.

As the student sees the system trying to prove Q and trying to prove R, they realize that the second half of their assertion had used an "and", instead of an "or". They modify their assertion, and re-validate. In this case their assertion is highlighted in green and the overall status of the proof is displayed as "Valid".

Usage Limitations. Two main limitations exist in this system. Firstly the logic language is restricted to that of propositional logic. Many scenarios requiring the use of formal proofs will be based in more powerful logics. Secondly, as the number of variables and clauses increase, the processing time for validation will reach a point where feedback can no longer be considered instant. Various mechanisms have been employed to mit-
igate this, as in the use of local contexts, but in cases where a single large proof is desired or required, the system will still have size limitations.

III. DESCRIPTION
The components of this system are depicted in Figure 2. All code in this implementation is written in JavaScript, and is organized as separate files each representing a JavaScript class. A single HTML page exists that creates an instance of the Canvas class, which it injects with an instance of the Context class, to represent the global context. As users create new partial proof widgets, new instances of the ProofWidget class are created and added to the instance of Canvas. Each proof widget, through its context also has access to an instance of the PropositionalLogicImplicationEngine, and PropositionalLogicEvaluator classes. The details of all components are described below, followed by a brief discussion of performance.

Canvas. The Canvas class represents a container for all partial proof widgets. When a partial proof widget is created it is registered with the canvas, and in turn receives a reference to the canvas. In this way, changes made to a partial proof can trigger updates in the rest of the system. The canvas thus acts as a common thread of communication between all widgets. In addition it provides functionality that exists at levels above an individual widget, such as organizing all partial proofs, displaying overall system status, and displaying overall system errors when requested. It is responsible for triggering the re-validation of globally importing widgets when a modification is made to a globally exporting widget.

Context. The context is responsible for holding the set of assumptions, and their implied statements. Assertions are performed through a context by asking it to validate itself with respect to a statement of assertion. A context holds a reference to an implication engine and an execution engine which allow it to expand on its assumptions, and conclude assertions respectively. A single context is created to be owned by the canvas, and thus act as the global context. In addition each partial proof widget is created with its own local context instance.

ProofWidget. The ProofWidget class represents the visual widget (text box, buttons, drop downs, draggable window, etc), as well as the data that is contained within. The ProofWidget can be in one of two states: active, or inactive. An active ProofWidget displays as a draggable window with a text box for input, and drop down menus for selecting the mode of import and export. An inactive ProofWidget displays as a simple text box, with modified text to show graphical representations of the logical connectives, and stylized/colored assertions to indicate their validity. Proof widgets communicate with the rest of the system via the canvas they were registered with.
Assume \{[P] \text{ and } [Q]\} or \{[R]\}.
Assert \{[P] \text{ or } [R]\} and \{[Q] \text{ or } [R]\}.

Assume \{[P] \text{ or } [R]\} and [Q].
Assert \{[P] \text{ or } [Q]\} and \{[R] \text{ or } [Q]\}.

Assume \{([P]A) \text{ and } [P]B\} and \{[Q]A \text{ and } [Q]B\} or \{[R]A \text{ and } [R]B\}.
Assert \{([P]A \text{ and } [P]B) \text{ or } ([R]A \text{ and } [R]B)\} and \{([Q]A \text{ and } [Q]B) \text{ or } ([R]A \text{ and } [R]B)\}.

Assume [Z].
Assume [Z] implies [G].
Assume [G] implies [K].
Assert [K].

Assume \{([P] \text{ and } [Q]\} \text{ and } \{[S] \text{ and } [T]\}\} implies [R].
Assume [P].
Assume [Q].
Assume [S] implies [T].
Assume [S].
Assert [R].

Figure 3: Example Input

PropLogicImplicationEngine. The propositional logical implication engine is the class that expands a set of assumptions to include its implied statements. This class has, as its main entry point, a function with the signature: \text{expandImplications(assumptions)}. This function and its subordinates will expand the set of assumptions provided, and keep accurate provenance information while doing so. Each assumption is expanded in the context of currently known statements. This is accomplished via a recursive descent algorithm that expands the current assumption if possible, and then calls itself recursively on the constituent parts of a complex statement if applicable. The expansion rules are modeled after the proof rules of propositional logic. For example given \text{\neg \neg P} the system will conclude \text{P} is also a valid statement, or given \text{P \rightarrow Q} and the fact (from other assumptions, or expansions) that \text{\neg Q}, it will conclude \text{\neg P}. In a first attempt at this logic the code would expand OR clauses using the rule which states that given \text{\phi \lor \psi} one can conclude the intersection of what ever can be concluded from assuming \text{\phi} and from separately assuming \text{\psi}. This proved to be unwieldy, however, as the expansion of such assumptions were often very large. Determining an artificial stopping point for the expansion proved to be either difficult to accurately implement, or easily implemented, but rendered the system incomplete (i.e. stopping at certain levels of recursion). To remedy this, the logic was moved into the evaluator class described below. In this location the code when checking \text{\phi \lor \psi} could then instead assume \text{\neg \phi} and ensure \text{\psi}, and vice versa - avoiding the need for large expansions.

PropLogicEvaluator. The Propisitional Logic Evaluator engine is responsible for evaluating assertions. It contains an entry point of a function \text{evaluate(assumptions, assertion)}. This function and it’s helper functions allow for the evaluation of a propositional logic statement given the assumptions of a context (which may include conclusions from the global context). Like the implication engine the evaluator operates as a recursive descent algorithm, evaluating a complex structure by first evaluating its component parts. For example in evaluating \text{\phi \land \psi} it will call itself recursively on \text{\phi} and on \text{\psi}. With few exceptions (for example the OR processing case mentioned above) this class is very straightforward. This class as with the PropLogicImplementationEngine, is intended to be pluggable in the sense that to use this system with a higher order logic one need only create implementations of these two classes that correspond to the same interfaces, and then inject these new versions instead of the existing instances into the created contexts. Since JavaScript does not natively support the notion of class interfaces (though it can be simulated to an extent), any future classes would simply need to contain methods matching the signatures of the entry points mentioned for PropLogicEvaluator and PropLogicImplicationEngine.

Performance Analysis. Performance in this system is adequate for many tasks. A single proof of one of De Morgan’s laws takes 15 milliseconds on average. A larger proof, as in figure 3, with a single proof box containing 14 variables, 11 basic or composite assumptions, and 5 assertions, executes in approximately 1200ms on average.

Clearly the runtime is not linear in the size of the input. The use of local contexts and multiple proof boxes...
can however allow for a near linear scaling in situations where these methods are acceptable.

IV. RESOURCES
The model for creating this system was the artifact automated assistant tool. Insight into potential use cases, layout, helpful features, etc were derived from use of artifact.

The following languages, tools, and frameworks were used directly in implementing the new system:

**JavaScript.** The entirety of the code base, with the exception of a single HTML page, is written in JavaScript.

**ExtJS.** ExtJS [3], a JavaScript library, was used to provide the user interface widget components. Elements such as draggable windows, toolbars, etc, which do not exist natively in HTML are provided as reusable components in ExtJS. ExtJS also helps to ensure a level of cross browser compatibility by accounting for browser differences in the component code, alleviating the developer from such concerns.

**JS/CC.** JS/CC [2] is a JavaScript parser generator, in the style of tools like Lex/Yacc, and ANTLR. It was used to generate a propositional logic parser (with extensions for Assume and Assert) from a grammar file in EBNF format.

V. REFERENCES