SSPARCy: A Software Integration Support and Design

Rationale Capture System

by

Quincy R. Scott

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

July 11, 2001

Copyright 2001 MIT. All rights reserved.
SSPARCy: A Software Integration Support and Design Rationale Capture System

by

Quincy R. Scott

Submitted to the
Department of Electrical Engineering and Computer Science

July 11, 2001

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The SSPARCy application was created to support the process of integrating software modules in an efficient fashion. SSPARCy performs automatic extraction of vital information from MATLAB source code. It then allows users to view this information, records the history of the states of the code, performs integration integrity checks and facilitates the capturing of design rationale associated with important code elements.
Acknowledgements:

I would like to thank all the people who have contributed to the development of the SSPARCy system and all those individuals who have supported me in the process. First, I would like to acknowledge the detailed feedback and guidance of the MIT SSPARC team, including Hugh Mcmanus, Joyce Warmkessel, Adam Ross, Myles Walton and Daniel Hastings. In particular, I would like to thank Bill Kaliardos for his interest and continued assistance in the development of SSPARCy.

I also wish to acknowledge the contributions of the other members of the MIT SSPARC IT team to the project. First, I would like to acknowledge Dr. Amar Gupta for his supervision of the project. I wish to thank Presley Cannady and Vida Hu for their work on implementation of GUI features. I especially would like to thank Shane Cruz who performed beyond the expectations of undergraduate work in his development of the majority of the GUI, a task that proved to be a very demanding undertaking.

Finally, I would like to sincerely thank all those individuals who have supported me in my work. I would like to thank my parents for their lifelong support. I would also like to thank Daniel Moon and Brian Wong for their support and camaraderie on many long nights of endless work.
Table of Contents:

1 Introduction .................................................................................................................. 7
  1.1 Research Task ........................................................................................................... 8
  1.2 Research Roles ......................................................................................................... 10

2 Background – Existing Tools ......................................................................................... 12
  2.1 Information Extraction ............................................................................................. 12
    2.1.1 MATLAB Debugger ............................................................................................. 12
    2.1.2 MATLAB Compiler ............................................................................................. 12
    2.1.3 DDD .................................................................................................................... 13
    2.1.4 WAVE ............................................................................................................... 14
  2.2 Software Integration .................................................................................................. 15
    2.2.1 SPIN .................................................................................................................... 15
    2.2.2 MSC, POGA, and TEMPLE .................................................................................. 16
    2.2.3 COOL:Spex ......................................................................................................... 16
    2.2.4 Catalysis .............................................................................................................. 18
    2.2.5 Boeing Tools ........................................................................................................ 18
    2.2.6 SSPARC Tools ..................................................................................................... 19
  2.3 Design Rationale Capture .......................................................................................... 24
    2.3.1 Meeting Minutes ................................................................................................. 24
    2.3.2 QuestMap ............................................................................................................ 25
    2.3.3 DRAMA ............................................................................................................... 26
    2.3.4 DRIM .................................................................................................................. 27

3 Description of SSPARCy ............................................................................................... 30
  3.1 Overview of System Components .............................................................................. 30
  3.2 Logical Representation ............................................................................................... 32
  3.3 Conceptual Representation ......................................................................................... 32
    3.3.1 Object Models ...................................................................................................... 33
    3.3.2 Highest-Level Conceptual Object Model ............................................................. 38
    3.3.3 Conceptual Entities ............................................................................................. 39
      3.3.3.1 Project ........................................................................................................... 39
      3.3.3.2 Function ....................................................................................................... 40
      3.3.3.3 Constant ....................................................................................................... 41
      3.3.3.4 Design Variable .......................................................................................... 41
      3.3.3.5 Error ........................................................................................................... 42
    3.3.4 Sub-Entity Interactions ......................................................................................... 42
    3.3.5 Function Interaction .............................................................................................. 44
    3.3.6 Entity Properties .................................................................................................. 46
    3.3.7 Complete Conceptual OM for a Given State ......................................................... 49
    3.3.8 Historic Conceptual Object Model ...................................................................... 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.8.1</td>
<td>Time</td>
<td>52</td>
</tr>
<tr>
<td>3.3.8.2</td>
<td>State</td>
<td>53</td>
</tr>
<tr>
<td>3.4</td>
<td>Code Representation</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td>SSPARCy Features</td>
<td>63</td>
</tr>
<tr>
<td>4.1</td>
<td>File Menu</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Project Menu</td>
<td>64</td>
</tr>
<tr>
<td>4.2.1</td>
<td>List View</td>
<td>64</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Historic View</td>
<td>67</td>
</tr>
<tr>
<td>4.2.3</td>
<td>N-Squared View</td>
<td>69</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Check Project</td>
<td>71</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Conclusion</td>
<td>74</td>
</tr>
<tr>
<td>5</td>
<td>Technical Implementation</td>
<td>76</td>
</tr>
<tr>
<td>5.1</td>
<td>Information Extraction</td>
<td>77</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Files</td>
<td>77</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Parsing Out Strings</td>
<td>78</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Parse Out Comments</td>
<td>85</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Functions</td>
<td>86</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Function Inputs</td>
<td>89</td>
</tr>
<tr>
<td>5.1.6</td>
<td>Function Outputs</td>
<td>90</td>
</tr>
<tr>
<td>5.1.7</td>
<td>Function Calls</td>
<td>90</td>
</tr>
<tr>
<td>5.1.8</td>
<td>Input Structure Variables</td>
<td>93</td>
</tr>
<tr>
<td>5.1.9</td>
<td>Output Structure Variable</td>
<td>99</td>
</tr>
<tr>
<td>5.1.10</td>
<td>Constants</td>
<td>100</td>
</tr>
<tr>
<td>5.1.11</td>
<td>Design Variables</td>
<td>103</td>
</tr>
<tr>
<td>5.2</td>
<td>Check Project</td>
<td>103</td>
</tr>
<tr>
<td>6</td>
<td>Comparison of SSPARCy to Other tools</td>
<td>107</td>
</tr>
<tr>
<td>6.1</td>
<td>Information Extraction</td>
<td>107</td>
</tr>
<tr>
<td>6.2</td>
<td>Software Integration</td>
<td>108</td>
</tr>
<tr>
<td>6.3</td>
<td>Design Rationale Capture</td>
<td>111</td>
</tr>
<tr>
<td>6.4</td>
<td>Comparison Summary</td>
<td>111</td>
</tr>
<tr>
<td>7</td>
<td>Future Research</td>
<td>113</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>116</td>
</tr>
<tr>
<td>9</td>
<td>References</td>
<td>117</td>
</tr>
<tr>
<td>10</td>
<td>Appendices</td>
<td>120</td>
</tr>
<tr>
<td>10.1</td>
<td>Back-End Architecture Code</td>
<td>120</td>
</tr>
<tr>
<td>10.1.1</td>
<td>ClientIO</td>
<td>120</td>
</tr>
<tr>
<td>10.1.2</td>
<td>Project</td>
<td>125</td>
</tr>
<tr>
<td>10.1.3</td>
<td>Function</td>
<td>146</td>
</tr>
<tr>
<td>10.1.4</td>
<td>Constant</td>
<td>165</td>
</tr>
<tr>
<td>10.1.5</td>
<td>DesignVar</td>
<td>166</td>
</tr>
<tr>
<td>10.1.6</td>
<td>Error</td>
<td>167</td>
</tr>
<tr>
<td>10.1.7</td>
<td>Call</td>
<td>168</td>
</tr>
<tr>
<td>10.1.8</td>
<td>Input</td>
<td>169</td>
</tr>
<tr>
<td>10.1.9</td>
<td>Output</td>
<td>171</td>
</tr>
<tr>
<td>10.1.10</td>
<td>InputStrucField</td>
<td>173</td>
</tr>
<tr>
<td>10.1.11</td>
<td>OutputStrucField</td>
<td>174</td>
</tr>
<tr>
<td>10.1.12</td>
<td>FunctionList</td>
<td>175</td>
</tr>
<tr>
<td>10.1.13</td>
<td>ConstantList</td>
<td>176</td>
</tr>
<tr>
<td>10.1.14</td>
<td>DesignVarList</td>
<td>177</td>
</tr>
<tr>
<td>10.1.15</td>
<td>ErrorList</td>
<td>178</td>
</tr>
<tr>
<td>10.1.16</td>
<td>CallList</td>
<td>179</td>
</tr>
<tr>
<td>10.1.17</td>
<td>InputList</td>
<td>180</td>
</tr>
<tr>
<td>10.1.18</td>
<td>OutputList</td>
<td>181</td>
</tr>
<tr>
<td>10.1.19</td>
<td>InputFieldsList</td>
<td>182</td>
</tr>
<tr>
<td>10.1.20</td>
<td>OutputFieldsList</td>
<td>183</td>
</tr>
<tr>
<td>10.1.21</td>
<td>Variable</td>
<td>184</td>
</tr>
<tr>
<td>10.1.22</td>
<td>TimeList</td>
<td>187</td>
</tr>
<tr>
<td>10.1.23</td>
<td>Constants</td>
<td>189</td>
</tr>
</tbody>
</table>

**10.2 GUI**

| 10.2.1 | Btos         | 190 |
| 10.2.2 | MainFrame    | 192 |
| 10.2.3 | Utilities    | 244 |
| 10.2.4 | FileSaver    | 245 |
| 10.2.5 | RationaleDialogGUI | 246 |
| 10.2.6 | SsparcTable  | 253 |
1 Introduction

Efficient software integration is a difficult goal to accomplish. Modern software engineering typically involves the production of large-scale applications. The inherent complexity of such multifaceted systems generally requires a top-down approach to constructing a suitable implementation. In this popular approach, the problem to be solved is clearly specified and then recursively divided into sub-problems which when combined are equivalent to the original problem. Each sub-problem is then assigned to an engineer (or small team of engineers) to develop functionality that will satisfy the particular sub-problem. When all of the sub-problems have been solved separately, the different sub-solutions are integrated together to form a complete solution to the original problem.

While developing an individual solution to a sub-problem is a concrete process, the task of integrating software modules lacks a standard practice. Integrating software typically involves ad-hoc practices and a centralized form of coordination. As the complexity of the original task grows, so does the number of sub-problems in which it is eventually divided. As the number of sub-problems increases, so does the number of sub-solutions that must be ultimately integrated together to form a complete solution. The difficulty of integrating software modules scales more than one-for-one with the number of modules due to the numerous dependencies and relationships between modules.

The problem of efficient software integration is rather intractable, not just due to the scale of the complexity, but because of the lack of a universal definition of what successful integration is. This reality implies that successful software integration is a
subjective determination. In creating a set of standards for a particular domain by which to compare the efficacy of software integration, integrators are providing the basis to begin development of tools to achieve these criteria. Once the desired results are known in a given context, techniques can be generated to aid and test the integration process.

Throughout the design and integration process, designers make crucial decisions. These decisions dramatically affect the overall behavior of a system. Each careful choice is motivated by some reason. This reason is the rationale underlying critical design choices. Generally, the rationale behind a decision is in harmony with accepted assumptions and established goals. However, often the rationale motivating a particular choice is not well documented, forgotten, or is based on invalid assumptions or a misinterpretation of goals. When such decisions are reviewed in hindsight without the aid of knowing the underlying rationale, it can be difficult to assess the validity of the action at the previous time and context.

This deficiency demonstrates the value of keeping a record of the reasoning associated with key design and integration decisions over time. Practices and tools to track this information are generally referred to under the heading of design rationale capture [2]. These tools can be leveraged to provide historical insight on important system choices and to assist in verifying the validity of design decisions.

1.1 Research Task

The Space Systems Architecture Research Consortium (SSPARC) is an initiative involving the Massachusetts Institute of Technology (MIT), the California Institute of Technology (Caltech), the Stanford University, the Naval War College (NWC), the Air
Force Research Laboratory (AFRL) at Hanscom Air Force Base, industry advisors and government sponsors [16]. One of the many goals of the consortium is to explore the application of information technology tools to improve the process of lean aerospace design [16]. At MIT, members of the Department of Aeronautics and Astronautics, Department of Electrical Engineering and Computer Science, and the Sloan School of Management participate in the SSPARC project [16].

As part of the SSPARC research effort, a Terrestrial Observer Swarm (TOS) system was designed. TOS is a satellite system designed to take measurements of key parameters of the Earth’s ionosphere. A crucial step in the design process of TOS is the development of a MATLAB simulation of the system [14]. This simulation is used to explore the efficacy of potential system architectures and to perform technical analysis of the design space [14].

The simulation is constructed by devising a series of MATLAB functions that interoperate with each other to model the behavior of a real satellite system [14]. The approach adopted by the SSPARC team for the development of the simulation code is to divide the entire task into discrete software modules based upon functionality. The interfaces to the different modules are carefully defined and then the modules are assigned to different designers to be developed separately. When the individual modules have been completed, they are all integrated together to produce the simulation.

The research task described in this thesis concerns the design and creation of information technology tools to support the process of software integration for the SSPARC team. More specifically, tools were developed to aid in the integration of the MATLAB simulation functions and to provide analysis of the simulation integration.
Through an iterative process with members of the SSPARC team, primarily the participating members from MIT, a set of desired objectives was identified involving automatic information extraction from MATLAB code, meaningful representation of this information, analysis of function integration and design rationale capture capabilities. Given these broad objectives, features were designed and refined for a comprehensive set of tools to support the software integration process carried out by the SSPARC team. These tools were aggregated to take the form of the SSPARCy system.

1.2 Research Roles

The SSPARCy application was developed entirely from scratch by the SSPARC Information Technology (IT) team with invaluable input from the other members of the SSPARC effort, especially those from MIT. Dr. Amar Gupta of the Sloan School of Management supervised the IT team. The design and implementation of the underlying SSPARCy architecture, information extraction methods, representation, integration analysis, design rationale capture, and historic simulation state capture were performed by Quincy Scott, a graduate student in the Department of Electrical Engineering and Computer Science of MIT. The creation of the SSPARCy graphical user interface (GUI) was led by Shane Cruz, an undergraduate student in the Department of Electrical Engineering and Computer Science of MIT, with supervision provided by Quincy Scott. Vida Hu was responsible for the production of the “N-Squared View” of the system and Presley Cannady implemented the attribute information dialog box. Jason Yeung implemented the “Historical View” feature of the GUI for the inputs and outputs of
functions, using the work of Shane Cruz as a guideline. Tara Sainath integrated the “N-Squared View” feature into the application’s GUI. Ms. Hu, Mr. Cannady, Mr. Yeung and Ms. Sainath are all undergraduate students in the Department of Electrical Engineering and Computer Science of MIT. Additionally, all worked under the direction of Quincy Scott with assistance and integration of GUI components performed by Shane Cruz.
2 Background – Existing Tools

2.1 Information Extraction

As part of the SSPARC IT research agenda, one key aspect is to provide tools to extract information from the MATLAB code files used to create the TOS simulations.

2.1.1 MATLAB Debugger

Recent versions of MATLAB are equipped with a built-in debugger [13]. This program parses MATLAB code and assists users in understanding its operations [13]. The debugger possesses knowledge of MATLAB syntax rules and is able to utilize these rules to extract information from raw code files. This information then enables it to assist users in debugging activities, such as monitoring variable values over time or step-by-step execution of code.

2.1.2 MATLAB Compiler

MATLAB is also equipped with a compiler. This compiler converts MATLAB source code into C or C++ source code [13]. The compiler possesses a strong knowledge of MATLAB syntax and the ability to successfully extract information from source code files. Additionally, this class of compilers is able to convert MATLAB functions into equivalent C/C++ library functions [13].
2.1.3 DDD

The DataDisplay Debugger (DDD) [26] is a graphical extension to existing source code debuggers (referred to as inferior debuggers in the DDD literature). DDD was developed by GNU, the open source initiative, and is freely available. DDD allows users to view source texts, runtime values of variables, relations between variables, and the evolution of values over time, and plots of large collections of values as well as their histories.
Figure 1: **Plots of Variable Values in DDD**

Figure 1 shows examples of the different graphical views of data available with DDD. DDD represents a major advancement in terms of visualization capabilities over existing command-line debuggers. This enhancement provides a convenient medium for integrators to have an alternative view of runtime values for system variables.

### 2.1.4 WAVE

WAVE [4] is an automatic, incremental induction algorithm for learning information extraction rules. The algorithm was developed by researchers of the Department of Computer Science at the University of Massachusetts, Amherst. WAVE learns from a stream of training instances, rather than from a predefined set. This permits it to learn new rules at any time. The latter property makes it especially well suited for interactive, dynamic applications. WAVE maintains a hierarchy of extraction rules based on the *covers* relation. The *covers* relation states that A covers B if and only if all constraints in A are also in B and any predictions made by A are made by B as well. Every training instance is placed in the hierarchy under the most specific rules in the hierarchy that cover it. Once placed in the hierarchy, the training instance is generalized, along with its siblings, to form new rules.

Algorithms such as WAVE could potentially be applied to the practice of extracting information from MATLAB source code.
2.2 Software Integration

2.2.1 SPIN

SPIN [6] is a generic verification system developed at the Bell Laboratories that enables the design and confirmation of asynchronous process systems. Its focus lies in verifying the correctness of process interactions. These interactions are specified in a program-like structure along with general correctness requirements in a formal notation. SPIN provides a methodology for connecting the structures modeling interactions and the formal correctness criteria for the purpose of verification. Unlike other formal methods, SPIN seeks to abstract implementation and language details, instead of focusing on basic functional requirements. SPIN is capable of only handling procedures with bounded growth. This allows it to deal with a finite set of correctness properties that are dependent upon the limited problem size. It also employs an optimized depth-first graph traversal method to ensure an exhaustive search of all system states. The algorithm terminates when a counterexample to a correctness requirement is discovered. SPIN checks requirements and behaviors for both their internal and their mutual consistency. The methodology assumes that designs are revised until its critical correctness properties are successfully proven.

SPIN requires significant overhead for system integrators by requiring the formal specification of correctness requirements and system components.
2.2.2 MSC, POGA, and TEMPLE

This set of tools [1] was developed at Bell Laboratories to aid the requirements engineer in capturing nearly all-behavioral requirements. MSCs (Message Sequence Charts) are a standardized representation for time sequence diagrams. MSCs capture the message exchanges of a system as a formal structure that can be used to perform automated consistency checks.

POGA allows for the graphical display of the dependencies between MSCs. POGA is a tool that facilitates the construction and analysis of directed label graphs. The combination of POGA and MSC can be used to detail the permissible, or required, sequence of events at the external interface to the system, while omitting the implementation details.

TEMPLE adds the functionality to search an MSC, or a collection of MSCs grouped in POGA graphs, for paths that correspond to a certain system behavior. Together, all three tools support a system to capture and analyze the requirements that define a new design. This functionality can be used to help verify that a system’s implementation satisfies design requirements.

2.2.3 COOL:Spex

COOL:Spex [29] is a component-modeling tool that enables domain analysis, component identification, component specification and the import/export of component specifications to implementation tools. It provides a series of diagramming tools that support the interface modeling aspects necessary to identify component specifications.
Unlike other tools, COOL:Spex is not technology dependent. It creates a distinction between specifications and implementations. A user is able to detail an interface, examine the interactions among interfaces and record steps in the development process. Once these interfaces have been specified, any component implementation tool that understands the interfaces can be used to generate code. One such tool that is compatible with COOL:Spex is COOL:Gen.

COOL:Spex and COOL:Gen were both created by Sterling Software as part of its ADD portfolio of tools. Additionally, an information repository, such as Microsoft Repository, can reside between COOL:Spex and COOL:Gen for storage and retrieval of information.

Figure 2: COOL:Spex and COOL:Gen Architecture

Source: [29]
2.2.4 Catalysis

Catalysis [28] is a methodology developed by Desmond D’Souza and Alan Wills for modeling and building systems from components and frameworks. It uses type models to capture the visible, external behavior of objects. Conformance between types is recorded as a movement from abstract to more detailed description of a system. Collaborations represent the interactions between components. Together, this captures a collection of actions involving several, typed objects serving defined roles. Identifiable patterns of organization or behavior are then recorded as frameworks.

Thus, Catalysis assumes that systems can be modeled as collections of interaction components, behavior can be analyzed by examining component interfaces, and that a formal notation is available for describing these interfaces.

2.2.5 Boeing Tools

Boeing uses a suite of tools to aid in the process of software integration. Boeing uses the applications FlyThru, Update, BuildIt, PigsFly, and TeleFly [27] to provide a variety of features. These tools offer interference checking, detection of motion anomalies, maintenance of documentation, and large scale manufacturing illustrations. Together, these applications are used for detailed analysis and design reviews of complex systems.
2.2.6 SSPARC Tools

Given the domain-specific nature of software integration processes, and the lack of any existing tools appropriate for the development process used by SSPARC, the SSPARC team decided to create its own set of proprietary tools [14, 15]. The current set of tools created by the SSPARC team supports the integration process by centralizing information access and performing rudimentary integrity checking.

The first tool is a template that is used to collect information regarding a function in the simulation.
Figure 4: **SSPARC Function Information Template**

Figure 4 shows the template used by the SSPARC development team to encapsulate information for a function. The template was developed in Microsoft Excel. The simulation integrators manually enter all function information and subsequent updates. The template contains the input and output structures for a function. It also lists the variables contained in the respective structures. In addition, different attribute information (units, creation date, author, definition, aliases) is presented for each variable and function.
Furthermore, several Visual Basic macros have been developed to operate in the Excel environment. These macros perform integration integrity checks by comparing a given function's information against other functions' information as entered into their respective template sheets. If an inconsistency occurs, the offending variable is highlighted in red, as shown in Figure 4.

In addition to function template sheets, the integrators have created similar templates for constants, design variables, and a listing of all functions in a simulation.

The SSPARC team has also developed a series of views to observe the flow of information between functions. The first is the N-Squared view.
The purpose of the N-Squared view is to show the dependencies and information flow between functions. If a function in column $j$ has a dependency on a function row $k$, then an $X$ is placed in row $k$, column $j$ of the table. The fact that row $j$ and column $j$ both represent the same function is handled by creating a diagonal of blacked out boxes from the upper left hand corner to the lower right corner. This divides the view into a lower
triangle and an upper triangle. Designers manipulate the order of the functions, always maintaining that if a function is in row j, then it is also in column j, in order to improve the flow of data between functions. Presently, all information for the N-Squared view is entered and updated manually. However, the N-Squared Excel template provides a convenient structure in which to maintain information for this type of analysis.

The SSPARC team has also developed an alternative view to better understand the flow of information between functions. This alternative view is shown in Figure 6.

![Module Flow Map](image)

**Figure 6:** SSPARC Function Information Flow Map
The SSPARC tools are the most tailored utilities to the domain-specific integration processes followed by the SSPARC team.

2.3 Design Rationale Capture

The difficulty with capturing high-level, intangible thought processes is that no adequate structure exists to characterize them. The interaction of creative individuals lacks an intuitive organizing form. The absence of this structure makes it extremely difficult to capture the reasoning underlying significant developments in a meaningful way [2].

2.3.1 Meeting Minutes

The oldest and most widely used form of design rationale capture practice is the use of meeting minutes. Minutes are a manually written log of chronologically ordered highlights from meetings involving designers. Minutes are notes that must be written down in some intelligible format. Furthermore, the creation of minutes generally requires the dedication of a single individual to the act of recording information. Minutes can be structured in an unlimited number of ways. Generally, they take the form of raw information that is very broadly organized by the recording individual in some format that is useful for information retrieval later. These formats vary widely and usually are specific to the user group keeping the minutes and the type of information. They have no inherent automation or analytical capabilities.
2.3.2 QuestMap

QuestMap [11] is a commercial tool for capturing rationale created by The Soft Bicycle Company. Its focus is on providing support for recording conceptual decision making processes. QuestMap attempts to model sequences of thought processes by offering templates that can be used to structure ideas. These ideas provide a convenient form and organization in which to record decisions and corresponding rationale.

Figure 7: QuestMap Design Rationale Capture

Figure 7 shows an example of using QuestMap to capture the decision processes relating to the marketing of a new product. QuestMap offers a series of icons and relations (color-coded arrows) to organize ideas. The icons are intended to represent concepts and
arrows correspond to relationships between concepts. QuestMap also provides hierarchical maps that are linked together to enable different layers of idea capture.

QuestMap does not offer any formal structures in which to organize ideas. Instead, it offers basic elements that are supposed to represent fundamental constituents of conceptual decision processes. The user is required to structure these elements in a meaningful and effective way. Additionally, QuestMap requires that a user enter all information manually. In this aspect, it is similar to meeting minutes in that it requires a designated individual to input information into the program, whether this occurs during a design meeting or after the fact.

2.3.3 DRAMA

DRAMA (Design RAtionale MAagement) [3] is a methodology and associated commercial software tool developed by QuantiSci for recording and managing design rationale. The tool was created based on previous work done as part of the KBDS (Knowledge Based Design System) developed at Edinburgh University. DRAMA employs a graphical notation for recording issues evaluated, arguments concerning positions and decisions made. This notation is based on the IBIS (Issue -Based Information Systems) approach. It is similar in presentation to that of QuestMap in that it supplies a series of icons that correspond to different categories of ideas and a set of relations between ideas. These basic elements can be used to construct decision trees that demonstrate the flow of ideas. Additionally, the application can be hyper linked with other applications.
2.3.4 DRIM

Design Recommendation and Intent Model (DRIM) [8] is a model for representing design rationale developed by the Department of Civil Engineering at MIT. It involves designers making proposals that satisfy a design intent. These proposals can be related to previous proposals. An artifact is the final product of this process. The designer presents a proposal that includes a recommendation and a justification. The recommendation introduces or modifies the components in a system.

The DRIM model is summarized in Figure 8.
DRIM provides a method by which design rationale information from multiple designers can be partially generated, stored and later retrieved by a computer system. It uses domain knowledge, design experiences from past occasions and interaction with designers to capture design rationale.

DRIM can be extended to the field of software module reusability. The result is a system called DRIMER. DRIMER allows users to search for previous designs by intent, find patterns, refine the search based on constraints, and attempt to incorporate code into...
a project. It utilizes the "patterns-by-intent" approach that refers to the process of selecting patterns based on their initial intents and then refining the choice of the pattern by specific constraints.

DRIM can be used to provide a model for classifying and explicitly representing design rationale. It requires significant overhead to explicitly mold the natural design process to fit the DRIM model. However, once this has been completed, analysis and improved understanding of the design process can be garnered.
3 Description of SSPARCy

3.1 Overview of System Components

The SSPARCy system provides efficient access to information regarding MATLAB source code files and analysis of their interactions. Figure 9 provides a high-level description of the major components of the system and their interactions with each other.

![High-Level System Interaction](image)

The Integrator is the user of the system. The Integrator interacts with the SSPARCy application by using the GUI to access and enter information. In addition, the Integrator creates MATLAB files and places them on the file system. The GUI makes requests for information from the back-end system logic. The back-end system performs
logical operations by analyzing the MATLAB files stored on the file system. In addition, the back-end can store and retrieve the state of a simulation on the file system.

The back-end system can be further decomposed into major modules by functionality.

Figure 10: **Back-End Architecture Components**

The back-end system consists of several components. The first is the information extraction functionality. This functionality enables the system to automatically extract information from MATLAB source code files. This extracted information is then used to create the state of the application. When the state changes, the old state is recorded. The
act of capturing previous states creates a history of the simulations. The entities of each state have attributes and design rationale associated with them. Furthermore, the state can be used to perform a series of logical verifications of the simulation’s integration.

### 3.2 Logical Representation

To present the information of the simulation to the user in a meaningful way and to provide useful analysis it is necessary for the application to construct a logical representation of the code. The application then offers users access to the information contained in this representation. In addition, it performs analysis based upon the organization of the information contained in the logical representation. The logical representation consists of the conceptual representation realized in the actual implementation (code object) representation.

### 3.3 Conceptual Representation

A crucial initial step in the development of the SSPARCy system was the formulation of the conceptual representation. The conceptual representation is a framework that captures and structures the concepts and data of interest contained in the source code files. The conceptual scheme implemented by SSPARCy was developed from the perspective of the SSPARC team. It attempts to organize the code in a way that is meaningful for the applications for which the SSPARC team uses MATLAB code. The scheme extracts those entities in the code that are important to the SSPARC team to observe and track over time.
It is important to realize that no single correct conceptualization of the system exists. In fact, many equally valid conceptualizations of the system are possible. The key is to choose a representation that includes all relevant information in an accurate, meaningful fashion and is efficient for the intended uses of the conceptual model. The version of the conceptualization described below satisfies these criteria. It is inclusive of all of the important information in the system that the SSPARC team wishes to monitor. It then organizes this information in a clear and useful way. Additionally, it is extensible, permitting later additions of other significant qualities and relationships of the system.

### 3.3.1 Object Models

For the purpose of expressing the conceptualization of a simulation, the paradigm of object models is used. Object models (OMs) [30] are employed to formally express a perception of a problem. They help a designer to figure out what information a system must retain, what state components are needed, how these relate to each other and what are the constraints of the problem. OMs capture the essence of the problem. They are precise expressions, but at the same time suppress irrelevant detail. By providing a meticulous conceptualization of the task at hand, OMs require the designer to reason in detail about the problem. By doing so, OMs help in identifying conceptual errors early in the development process. Additionally, a well-formed OM facilitates the implementation of the corresponding object-oriented code.
Many different notations exist for object models. The one used in this document is called Alloy. Alloy is an object modeling language developed at MIT [30]. It is a simpler version of the popular UML notation.

In Alloy, a node represents a set of objects. Each node is considered to consist of all objects of a certain type. This is called a set. For instance, if an OM were modeling a family tree, a node entitled Man would represent the set of all men. Sets can be fixed, meaning that no new objects are added or removed from the set. In the family tree example, a fixed set for Man would imply that no man could die or be born. Sets can also be static. A static set means that no existing object can move in or out of the set. Continuing with the family tree example, if a set for Man and a set for Woman exist, both would be static implying that no Man could become a Woman and no Woman could become a Man (although this is not always true, it is generally a good approximation). The notation for sets, fixed and static are shown below in Figure 11.
OMs also contain arcs. Arcs are relations between objects or are subset relationships. Arcs with an open arrowhead denote a relation between sets of objects. Arcs with an empty triangle arrowhead represent subset relations. Arcs with a filled triangle arrowhead represent an exhaustive subset relation, meaning that the subsets connected by the arc are all of the possible subsets. Figure 12 depicts the notation for arcs using the family example.
Figure 12: **Arc Notation for Object Models**

Figure 12 demonstrates how a Man is a subset of a Person. Also, it shows how Man and Woman are the only possible subsets of Person. In addition, the figure shows an example of the wife relation between a Man and a Woman. Notice that the transpose of the relation is also valid, meaning that if a wife relation exists from a Man to a Woman, then a husband relation exists form the same Woman to the same Man.

Mutability of relationships is also an important concept in OMs. Mutability concerns the ability of a relation between objects to change. For instance, a Man can share the wife relation with more than one Woman in the set of Woman over time, but never more than one at a particular time. This is different from an immutable relation in which the two related objects cannot share that relation with any other object. For example, the mother relation from Son to Mother is immutable. Mutability in a relation...
is indicated by a hash mark crossing the arc. For the relation A to B, a hatch line on the left end of the arc indicates that the same A maps to B, during its life. A hatch line on the right side signifies that during its lifetime, each A maps to the same B. For instance, if a Man could only ever marry one Woman (for whatever reason), but that Woman can divorce him and marry another Man. In this case, a hash mark would cross the arc near the end where the Woman object is.

OM relations also express multiplicity. Figure 13 shows the symbols used in Alloy to express multiplicity.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0 or more (default)</td>
</tr>
<tr>
<td>+</td>
<td>1 or more</td>
</tr>
<tr>
<td>?</td>
<td>0 or more</td>
</tr>
<tr>
<td>!</td>
<td>Exactly 1</td>
</tr>
</tbody>
</table>

Figure 13: **Multiplicity Notation in Object Models**

If a relation from A to B exists such that each A maps to exactly one B and every B maps to zero or more A objects, then a “!” symbol would be placed near the arc on the end close B and a “*” would be placed on the end of the arc near A.

With these basic elements of the Alloy notation, very expressive OMs can be created to aid in correctly conceptualizing the task of modeling a simulation.
3.3.2 Highest-Level Conceptual Object Model

The conceptualization of SSPARC simulations that is implemented in the SSPARCy application can be understood by first examining the highest-level of the representation. Figure 14 is a conceptual object model (OM) depicting the highest-level of the conceptual representation of the MATLAB source code comprising a simulation.

![Diagram of Conceptual Object Model](image)

**Figure 14:** Conceptual Object Model of a Project at a Moment in Time
3.3.3 Conceptual Entities

3.3.3.1 Project

For the conceptual representation used in SSPARCy, the highest-level entity is the Project. A Project corresponds to all of the code files used in a simulation of a single mission. A Project is an all-inclusive entity encompassing all simulations of components and subcomponents of a complete system. A Project does not interact with other Projects directly. The object model shows the relationship between a Project and the entities that it directly contains. These entities may in turn contain other entities and thus cause Project to contain them indirectly. However, the Functions, Constants, Design Variables and Errors it consists of collectively define a Project. A Project contains all of these entities and thus the diagram depicts a “contains” relationship between Project and all of its sub-entities.

Figure 14 omits other entities that are related to a Project. These entities are descriptive of a Project, but do not define a Project. Thus, they have not been included in the highest-level OM, in order to center attention on the entities and relationships that define a Project. Additionally, relationships between the immediate sub-entities of Project have been omitted momentarily for the sake of clarity. Only relationships between Project and the immediate sub-entities that comprise it are shown.

Note that all sub-entities of Project are only valid representations for a given state of the system. If any entity in the Project changes, this results in a change to the Project and is counted as a time step as well as a new time period. From one time period to the
next, any entity may change or remain unchanged. Figure 14 is a snapshot of the conceptual model at the highest-level for a particular time period.

### 3.3.3.2 Function

The first immediate sub-entity that a Project contains is a Function. A Function corresponds to a globally visible MATLAB function (not any locally visible sub-functions). Each MATLAB function is defined in its own unique file ending in a “.m” extension. MATLAB functions take fields and structures as inputs, perform some operations utilizing the inputs, and produce fields and structures as outputs. A definition of fields and structures are provided in section 3.3.5 “Function Interaction” below.

MATLAB functions can be divided into three categories based upon their inputs and outputs. A Function takes fields or structures as inputs and produces them as outputs. Stricter requirements on the inputs and outputs of a Function allow some Functions to be further classified into one of two subclasses. If a Function does not take any inputs, then it is called a Script. If a Function produces a structure as an output, then it is a Module. Thus, all MATLAB functions are represented in the conceptual scheme as a Function or further refined to be a Script or Module.

Note that for the purposes of the SSPARC team, the distinction between a Script, Module, and Function is not vital for the purpose of information tracking or analysis. As a consequence, SSPARCy does not represent or treat Functions, Scripts, or Modules differently. However, for the sake of completeness, and to facilitate the potential future implementation of specific features, the notions of Scripts and Modules are included in
the conceptual scheme. Since each MATLAB code file corresponds to exactly one Function, a simulation must contain at least one Function. Thus, the relationship between a Project to Functions is one-to-one or one-to-many.

3.3.3.3 Constant

The next sub-entity contained by a Project is a Constant. A Constant represents a globally shared constant value. Most simulations include a file, typically named “Constants.m”, which defines a set of constant values. This set of constant values occurs in the MATLAB code as a structure consisting of constant fields. Each of these fields corresponds to a Constant in SSPARCy’s conceptual representation. A Constant represents the value of a constant and its associated information at an instance in time. The relationship of Project containing Constant is one-to-zero or one-to-many. This implies that a simulation (represented by a Project) can contain anywhere from zero or more constant fields. Projects typically contain many Constants.

3.3.3.4 Design Variable

A Project also contains Design Variables as immediate sub-entities. A Design Variable signifies a design variable in a simulation. A simulation will usually have several key variables, generally specified in a “Design.m” file. The purpose of the simulation is to observe the behavior of the simulated system to different combinations of values of these key design variables. A Design Variable captures the value and relevant information for a design variable at a given moment in time. A simulation typically
consists of several design variables, although it can (in a very uninteresting simulation) contain none. Thus, the relationship of a Project to Design Variables is one-to-zero or one-to-more.

3.3.3.5 Error

The final immediate sub-entity contained in a Project is an Error. Errors represent errors noticed when SSPARCy performs an integration check of the system. These errors result from invalid interactions between Design Variables, Constants, and Functions. An Error corresponds to an error in the simulation at a given time. A simulation may have zero or more errors.

3.3.4 Sub-Entity Interactions

The next addition to the conceptual representation are the relationships between the immediate sub-entities of Project.
Figure 15:  Conceptual Object Model of Sub-Entity Interaction at a Moment in Time

Note that the subclasses of Function (Script and Module) have been omitted from Figure 15 for the sake of clarity. The interactions of all Functions (including Scripts and Modules) are the same with regard to the sub-entities of Project.

Errors are the result of incorrect interactions among other sub-entities. Each Error can have exactly one cause. An Error can be caused by any of the sub-entities. Thus, an Error has a zero or one relationship with all of the sub-entities, meaning that any sub-entity can cause exactly one Error, but does not necessarily produce an Error.

Functions perform operations. To that end, Functions may call other Functions to perform operations and use the resulting output. Functions, therefore, have a relationship with each other. A Function may call zero or more Functions. All Functions must be called by at least one other Function. If not, the Function is not being used as part of the simulation. One exception to this relationship exists. The first Function called is not
called by a Function that is part of the simulation. Instead, it is called by some source external to the simulation, such as a user or another program.

Functions use Constants and Design Variables as part of the operations they perform. Functions may use any number of Constants or Design Variables. A Function is capable of producing numerous errors. Thus, Functions have a zero or more relationship with Errors.

Each Constant and Design Variable can cause a single Error. The relationship with Error is therefore zero or one.

### 3.3.5 Function Interaction

The purpose of a project simulation is to observe how different inputs, configuration of functions and architectures produce different outputs. The key to creating these valuable outputs is not just the operations of the individual functions comprising the system, but the aggregate interaction of functions.

In MATLAB, this collective interaction takes the form of information exchange. Functions can accept inputs, use them to perform meaningful operations, and then create a useful output. Interactions among functions occur when the output of one function is passed as the input to another function or when a function makes a call to another function (passing it input parameters) and uses the resulting output.
Figure 16: Conceptual Object Model of Function Data Interaction at a Moment in Time

Figure 16 depicts the interaction of the functions and related data of a system at a point in time. The interaction of a function with the types of data in the system determines exactly the class of the function.

The system consists of two types of inputs and outputs that are exchanged by functions. The first is a Field. A Field is basically a variable that represents some single value. The other kind of data is a Structure. A Structure is a logical collection of related variables. It contains many variables that are passed to and from functions together. The variables within a Structure can be considered to be equivalent to Fields. However, the variables of Structures are not exchanged between functions individually, but only
collectively with the rest of the variables of the Structure. Fields are passed independently among functions.

Functions can be divided into three categories, Functions, Modules and Scripts, based upon the types of data they take as inputs and the type they produce as outputs. A function that is classified strictly as a Function can take zero or more Fields as inputs. It also can take zero or more Structures as inputs. A Function, though, will only produce zero or more Fields as outputs. Modules can accept zero or more Fields as well as zero or more Structures as inputs. Modules can also produce zero or more Fields as outputs, just like Functions. The distinguishing characteristic of Modules is that in addition to Fields, Modules can produce zero or more Structures as outputs. Similarly, Scripts can produce Fields and Structures as outputs. The distinctive quality of Scripts is that they do not accept any inputs. In this manner, Scripts interact differently with the rest of the system than Functions or Modules. Scripts only produce data to be used by other functions. In general, they produce the same data regardless of the state of the rest of the system. As a reasonable consequence, Scripts are generally used to create important Structures or Fields for use by the remainder of the project. Thus, it is the types of data taken as inputs and outputs that defines the classification of a function and determines the nature of its interaction with the project.

3.3.6 Entity Properties

Each entity at all levels of the conceptual model of the system has a variety of properties associated with it.
Figure 17: **Conceptual Object Model of Entity Properties at a Moment in Time**

Figure 17 shows a generic entity and all of its associated properties. The properties shown in Figure 17 are all of the possible properties of an entity that have been identified by the SSPARC team as important to track and record in the SSPARCy tool. Name is simply the name given to the entity. Each entity can and must have only one name. Value is the current value of the entity. An entity can only have one value; however, some entities do not have a logical equivalent of a value. Date is the date upon which the entity was first created and incorporated into the system. Author is the author or authors responsible for creating the entity. Alias is another name by which this entity might be referred to. An entity can have zero or more aliases. Rationale is the reasoning for the creation and inclusion of this entity in the project. Each entity has a purpose or reason for being part of the simulation even though this rationale is not often explicitly
stated. Valid Range is the range of values that are permissible for the value of this entity to assume. Valid Range only makes sense in the context of entities that have values. Each entity with a value has a valid range for that value. In fact, an entity may have several valid ranges for the value it represents. Units correspond to the units of the value associated with the entity.

The entities in the system modeled by Entity are Functions, Modules, Scripts, Constants, Design Variables, Errors, Structures, Fields, and Project. However, not all of the properties above apply for each type of entity. Instead, the above properties are the superset of all possible attributes of any entity.

A Project will have a Name, Date, Author, Aliases, and Rationale. A Project does not have a value that it represents. Without a value, it also does not have units or a valid range. A Function, Module and Script all have names, creation dates, authors, aliases and rationale. Functions and its subclass entities do not correspond to values and thus do not possess units or valid ranges. Constants, Design Variables, and Fields all have names, creation dates, authors, aliases and rationales. The author for a Constant, Design Variable, Field or Structure is typically the person who chose to include the entity in the system. Additionally, Constants, Design Variables and Fields all represent values, thus they each have a value, units and valid ranges.

Structures are similar to Fields in that they too have a name, creation date, author, aliases and rationale. However, unlike a Field, a Structure does not have a value or the related properties of units and valid range. This difference exists because while a Field corresponds to one-for-one with a value, a Structure represents a collection of values, but no one specific value.
Finally, an Error is a rather unique entity in the system. It is the only type of entity that was not intentionally included in the system. Since it was not intended to be part of the system, it is not given a unique and identifying name by a creator for the purpose of recognition by users, or any aliases. Although an Error has an attributable source, it does not have an author in the sense of a person or persons responsible for including the entity in the simulation. Since no one incorporated an error into the system, it does not have a rationale or reason why a user included it. Instead it has a source or cause to which it can be attributed.

3.3.7 Complete Conceptual OM for a Given State

Using the above elements, a complete conceptual picture of how a simulation looks at any given moment in time can be created.
Figure 18: Complete Conceptual Object Model of System at a Moment in Time
Figure 18 is the aggregation of figures 14, 15, 16 and 17. It depicts the major entities of the system, their relationships to each other and an example of their attributes. For illustrative purposes, only the attributes of Project are included in the diagram.

This diagram represents the complete conceptualization of a simulation at a single moment in time. It includes all of the important entities and relationships that the SSPARCy tool will be expected to extract, capture and analyze for any given state of the simulation.

3.3.8 Historic Conceptual Object Model

The present conceptual model represents the components and interactions of a simulation for any state. This information can be used in aiding an individual in understanding the condition of a simulation at a given point in time. However, this perception possesses a limitation. Real simulations are not static, but instead evolve over time. As successive simulations are conducted, information is learned that prompts the designers of a project to change the elements and configuration of a simulation. These changes, although made to the same project, produce a new state for the project. Typically this new state will closely resemble the previous state. However, dramatic modifications can be made to a project resulting in a new state that bears little resemblance to its previous form.

When analyzing system architectures and choosing new potential configurations, analysis of previous states of a simulation are essential. Examining a single state of a system is often not sufficient to perform a thorough and effective analysis. Instead, several states must be compared with each other and the evolution of results must be
observed. Knowledge of previous configurations places the current state in context and permits evaluation of the system relative to other architectures, based on objective predefined criteria. This combination of comparisons allows the designer to see how well the current architecture satisfies the customer's criteria as well as how it compares to other potential architectures.

Viewing the history of states for a simulation is the result of examining the entities and relationships that comprised each state at each moment in time. Thus, it is essential to record the states of each entity and its relationships over time. To incorporate the perception of changes in states into the conceptualization of the system, the notion of history is introduced. Each major type of entity will have a history associated with it. This history will capture which entities comprised the major categories of entities at any moment in time. These entities will contain their state and relationships to other entities at the time of their creation.

3.3.8.1 Time

A specific time period, in the context of the conceptual model, represents a given state of the aggregate simulation. If the state of the simulation changes, then the new state corresponds to a new time period. Thus, the evolution of time represents the evolution of simulation states.
A simulation state is the aggregate state of the entities that comprise it. If the state of any entity changes, then this dictates a new aggregate simulation state. The state of an entity is not simply the attributes and relationships of the entity. Although all of these are associated with an entity, a change in them does not necessarily represent a change in state. The modification to an entity that corresponds to a change in state is dependent upon what is considered to be the important qualities and relationships of an entity. An entity can change in some fashion, but if the change is not significant, then the state of the entity may be considered to be unchanged. Therefore, the criteria for the definition of state for each entity in the simulation must be delineated.

A Constant or Design Variable represents a value. A change in this value corresponds to a change in the state of a Constant or Design Variable. An Error does not ever change state. It either exists or does not. While the underlying error represented by an Error is present in the simulation, then the Error exists. When the error is resolved, the equivalent Error is no longer part of the system. A Function (as well as a Module and a Script) represents a set of operations performed on inputs producing specific outputs. It therefore is characterized by its signature, which consists of its name, the name of its inputs and the name of its outputs. A change in this signature is a change in state and is indicative of a modification on the operations that the function performs. A Structure is characterized by the variables it contains and its name. A Field is defined by its name. A Project is a collection of Constants, Design Variables, Functions and the resulting
Errors. A change in the membership of this collection constitutes a change in the Project as a whole.

With this definition of time and state, a complete conceptual model of a simulation can be constructed.

Figure 19: Conceptual Object Model of the History of a Project
Figure 19 depicts the conceptual history relationship for a project. Note that only relationships concerning histories and inclusion are included in Figure 19.

A History is a collection of groupings of a certain type of entity. Each grouping is a set of entities that are associated with the entity containing the history at a given moment in time. If a change occurs in this group, a new group is constructed. The sequence of these groups over time forms a collection that corresponds to an entity history.

A Project contains exactly one Function History that represents the history of all the Functions used in the project. If a function is added or removed from the project, then the grouping of Functions changes and a new one is created. This causes the Function History to be updated.

A Project also contains one Constant History that illustrates the history of all Constants and their corresponding values. If a Constant is added or removed from the Project, then the Constant History is updated. Also, if the value of a Constant changes, the history is updated. Since Constants represent values, it is desirable to record the evolution of values for a Constant as well as which Constants are being employed.

Similarly, a Design Variable History is associated with the Project. The Design Variable History shows the Design Variables used in the project over time as well as their associated values. If a Design Variable is included or removed from the project, the history is updated. Also, just as with Constants, Design Variables correspond to values that are initially configured. If these values change, it dramatically alters the behavior of the simulation. Thus, it is advantageous to capture the sequence of values for a Design
Variable. This implies that the Design Variable History must be updated every time the 
value of a Design Variable is modified.

Projects also include an Error History. This history tracks the Errors that are 
produced by a simulation at any given time. Errors can only be created or resolved. 
Whenever either of these actions occurs, the group of Errors currently present in the 
simulation changes. This change causes an update of the Error History.

In addition to the Project, other entities contain histories. Each Function has an 
Input History and an Output History. Inputs and outputs to a Function take the form of 
Structures or Fields. The Input History represents the history of the inputs to a Function. 
If the group of inputs to a Function changes, this invokes an update of the Input History. 
Similarly, the Output History stands for the historic accounts of the outputs produced by 
a Function. If the group of outputs resulting from a Function is altered, then the Output 
History records this transformation. Obviously, Scripts do not possess Input Histories 
since they do not accept inputs. Alternatively, the Input History of a Script can be 
considered to be unchanging and null. Also, the Output History of a Function contains 
only Fields, while the Output History of a Module always contains at least one Structure.

Structures are groupings of variables, which are functionally equivalent to Fields. 
If the group of variables that a Structure contains is modified, this symbolizes a change in 
state for the Structure. This mutation is noted in the Variable History.

It is interesting to observe the flow of time for a simulation. In the strictest sense, 
a Project changes state (advances one unit time) whenever any of the Histories it 
recursively contains (meaning any Histories it directly contains or any Histories 
contained by a sub-entity recorded by the Histories the Project contains, or a sub-entity of
the sub-entity and so on) are modified. However, the real value in recording histories is not simply observing the changes in a Project. The real value arises from explaining the evolution of transformations in architectures. This insight is garnered from careful analysis of the progression of the entities contained in the Project, or recursively contained in the Project. Therefore, the Histories encapsulated in the Project provide the explanation for the evolution of the simulation. The existence of the meaningful history occurring in the recursive sub-entities of a Project is evident in the fact that no history for a Project is present. A Project History does not make sense in the context of a Project described so far. A change in a Project is a macro-level event arising solely from micro-level events. Since our research interest lies in the cause of the change in a Project, not simply the fact that it has changed, it is appropriate to only keep the histories of those entities that elicit the macro-level modification.

Another interesting observation is the evolution of time in each History. Since the Histories are not dependent on each other, they each update independently. This means that when one History updates, another History does not necessarily change. Since a change in a history defines a time step, this implies that each History has its own sense of time.
3.4 Code Representation

The full specification of the conceptual object model, including time, leads directly to the design of the code object model. The code object model is a representation of the code of the system. It details the architecture and organization of the code objects that comprise SSPARCy. See Figure 20.
Figure 20: Code Object Model for SSPARCy
The code object model shown in Figure 20 follows the same rules as described for the conceptual object model in the previous section. The system architecture, as shown in the code object model, can be divided into three major sections. The first is the GUI. The GUI interacts with the user and the second major portion, the ClientIO object. The ClientIO object represents a consistent interface that the GUI can deal with. The GUI makes requests to the ClientIO, which then returns the appropriate information to the GUI to be displayed to the user. The ClientIO creates a point of abstraction for the GUI. The GUI does not have to be concerned with the architecture of the system or where the information it receives comes from. It only needs to know how to process the information returned by the ClientIO. This enables the insertion of network functionality, or other potential extensions to the system, behind the ClientIO interface.

The ClientIO interacts with the final major section of the system, the back-end architecture. The back-end architecture consists of the Project object and all objects beneath it in the hierarchy shown in the code OM. Every object in the back-end is either a list or an entity. A list is a collection of entities of a certain type at a given time. Entities themselves can contain lists, which are a collection of other entities associated with the particular entity.

All lists have the same basic functionality. To reduce code replication of this functionality in each type of list, all lists in the back-end are subclasses of a generic class called TimeList (shown in appendix 10.1.21). TimeList simply provides the rudimentary functionality that all lists need, such as adding/retrieving an entity in the list and testing if a certain entity is a member of the list. FunctionList, ConstantList, DesignVarList,
ErrorList, CallList, InputList, OutputList, InputFieldList, and OutputFieldList are all subclasses of TimeList.

Similarly, all entities share similar properties and functionality. To aid in this uniformity, all entities are subclasses of a generic class called Variable (shown in appendix 10.1.20). Variable provides basic properties and methods to access and modify those properties. Project, Function, Constant, DesignVar, Error, Input, Output, InputStrucField, and OutputStrucField are all subclasses of Variable.

The root object of the back-end architecture is the Project object. It contains a collection of FunctionLists, ConstantLists, DesignVarLists, and ErrorLists. Since each list represents the state of a certain aspect of the simulation at a moment in time, a collection of any type of list corresponds to a history of that aspect. For instance, the collection of ConstantLists kept by Project is the history of the constants in the simulation since each respective ConstantList is a collection of the constants at any given point in time. Thus, the Project object directly keeps a history of functions, constants, design variables, and errors.

Each list is a collection of a certain kind of entity. FunctionList is a collection of Functions. ConstantList is a collection of Constants. DesignVarList is a collection of DesignVars. ErrorList is a collection of Errors.

Function objects also maintain histories of other entities that relate to functions. A Function contains a collection of InputLists, OutputLists, and CallLists. Thus, a Function keeps a history of its inputs, outputs and the other functions it makes a call to. CallLists contain Calls, which represent a call to another function in the simulation. InputLists contain Inputs, which is a set of any field/structure that is an input to the given
function. Similarly, an OutputList contains Outputs, which are any field/structure that is an output from the given function.

Inputs and Outputs represent fields or structures. If they correspond to a structure, then it is necessary to record a history of the fields that are contained in that structure. To that end, Inputs keep a collection of InputFieldsLists and Outputs keep a collection of OutputFieldsLists. InputFieldsLists contain InputStrucField objects, which represent a field of an input structure. Similarly, OutputFieldsLists contain OutputStrucField objects, which represent a field of an output structure.

Thus, the back-end architecture of SSPARCy is a hierarchy of entities and histories of lists of entities as specified in the code OM of Figure 20.
4 SSPARCy Features

The SSPARCy tool is primarily organized into a series of menus. The menus follow the scheme of the conceptual model, in that the menus offer access to functionality for different levels of the conceptual OM.

Figure 21: SSPARCy Main GUI

Figure 21 is a screenshot of the central screen for the graphical user interface (GUI) of the SSPARCy application.
4.1 File Menu

The file menu allows users to create a new project. A user specifies a name for the project and provides the directory on the local machine where the MATLAB files for the simulation are located (see section 5.1.1 Files, for a discussion of how the application processes files). Note that the tool assumes that all relevant code files for the project exist only in the specified directory. Once a project has been created, the file menu can be used to save the state of the representation of the simulation created by the tool to the file system. This permits more permanent storage and allows files corresponding to projects to be transferred from one computer to another. A previous project that has been saved also can be opened using the file menu. The current project can also be closed so that a new one can be created or opened.

4.2 Project Menu

The project menu allows users to access and enter information regarding the major entities of a simulation.

4.2.1 List View

The list view provides a listing of important entities of a project and related information. Presently, list views are available for functions, constants and design
variables. Also, list views have been implemented for viewing the inputs and outputs of a function, as well as the fields of input and output structures used in a function.

![List View](image.png)

**Figure 22: List View**

Figure 22 shows a screenshot for the list view of the constants for the BTOS simulation. This picture only includes a subset of the entire set of constants in BTOS. Entities (constants, design variables, functions) have additional attributes associated with them (see section 3.3.6 Entity Properties). However, only the name and value are displayed by default because these attributes are the only pieces of information that can be automatically extracted from the MATLAB source code. Additional attributes can be entered into the system by selecting a row and clicking the ‘Edit Info’ button. The latter operation brings up the dialog box shown in Figure 23.
The dialog box allows users to enter information for the attributes of units, valid range, author, aliases and rationale. If the rationale requires additional space than that presented by default, the "More" button can be pressed and a separate expanded text entry area will be displayed. Notice that the name and value attributes are grayed out. This indicates that they cannot be modified since they are automatically extracted from the underlying MATLAB code files. The additional attributes of an entity can be displayed by selecting the desired attribute from the sequence of check boxes in the center of the screen in Figure 24.
Figure 24: List View Toggle Attributes

As Figure 24 shows, a new column is displayed for each attribute that is checked. This capability enables any combination of attributes to be displayed. Note that if the attribute information for an entity has not been manually entered, or is not applicable for this particular entity, then it will be displayed as an empty cell. Also, observe that the date is automatically generated by the system and marks the time of the creation of the representation of the entity. If a user wishes to view the rationale for a particular entity, the user must select the corresponding row and then click the "View Rationale" button. A dialog window containing the expanded text of the rationale will be displayed.

4.2.2 Historic View

A powerful feature of the SSPARCy tool is the ability to store and view the history of the states of the entities of a simulation. Each time the code for any entity of the simulation is modified, the user must manually inform the SSPARCy application (by selecting the appropriate update choice from the Project Menu) that an update of state has occurred. The user does not need to detail the alteration of state. The systems will re-examine the code and determine if any meaningful changes have occurred (see section...
3.3.8.2 State, for a description of what changes are considered important for which entities). If a significant modification has taken place, the system records the old state for the modified entity and then updates itself to reflect the current state.

![Design Variables History Table]

**Figure 25:** Historic View

Figure 25 demonstrates the capabilities of the Historic View. In this screenshot, the history of the design variables for the simulation is shown over four time periods. Changes in the state are highlighted in a convenient and meaningful manner. If an existing entity changes value, then the cell in which the change occurs is highlighted in red. If an entity is removed from the simulation, then the deletion is noted by coloring the first empty cell gray. If a new entity is added to the simulation, the cell in the new time step is colored green. Thus, in the above screenshot the design variables “foo” and “swarms_per_plane” have changed values from time period 1 to time 2. The variable “processing” has been deleted in time 3 and the variable “new” has been added in time 4. By highlighting and color-coding changes, users can quickly scan the history of an entity.
and ascertain how a variable has evolved over time. Users can then select a cell in the historic view and view the rationale associated with the change in state.

This capability provides a powerful new dimension of being able to access design rationale history. A user can observe the progression of the state of individual entities or groups of entities over time. Additionally, the system records the reasoning associated with important changes in the simulation. The Historic View is the primary interface through which users utilize the design rationale capture capabilities of the SSPARCy tool. The Historic View is presently available for design variables, constants, and functions. It also includes Historic Views for errors, inputs, outputs, and the fields of input and output structures.

4.2.3 N-Squared View

Another way to view the functions and their interactions is an N-squared diagram.

Figure 26: N-Squared View

Figure 26 shows a screenshot of the N-squared view. The N-squared view consists of a table in which the column and row names are the names of functions in the simulation. If
column \( j \) has function name ‘foo’, then row \( j \) will also have function name ‘foo’. The purpose of the N-squared view is to show the dependencies between functions. If a function ‘foo’ is dependent on function ‘bar’, then an X is placed in the intersection of the row named ‘foo’ and the column ‘bar’. For the intersection of a row and column with the same name, the word ‘same’ is placed in the cell.

There exist two types of dependencies in simulations. The first is a function call dependency. If in the code of function ‘foo’ a call is made to function ‘bar’, then a dependency exists from ‘foo’ to ‘bar’. This would result in an X being placed in the intersection of row ‘foo’ and column ‘bar’. The second kind of dependency occurs when one function has the output of another function passed to it as input. For instance, if function ‘bar’ outputs a structure, ‘BAR’, and ‘foo’ takes ‘BAR’ as an input, then a dependency would exist from ‘foo’ to ‘bar’. This would also result in an X being placed in the intersecting cell of row ‘foo’ and column ‘bar’. Currently, the only dependency of the second type recognized by the SSPARCY system is if a function takes the output of the Design or Constant scripts as input. This limitation arises because of the lack of an adhered to standard for the naming of outputs and inputs as well as the absence of a MATLAB enforced practice. It is assumed that the output of the Design script is a structure named DESIGN and that the output of the Constant script is a structure named CONSTANTS. Thus, if a function takes DESIGN as input, it is assumed that it is dependent on the Design script and if it takes CONSTANTS as input, it is assumed to have a dependency with the Constants script.

N-squared diagrams are used by the SSPRAC and other development teams to view the dependencies and information flow between functions. The fact that row \( j \) and
column $j$ both represent the same function causes a diagonal of the word “same” to occur from the upper left hand corner to the lower right. This divides the view into a lower and an upper triangle. A designer can manipulate the order of the functions, always maintaining that if a function is in row $j$, then it is also in column $j$, in order to improve the flow of data between functions. To that end, the N-squared view allows users to drag columns to any desired position. The corresponding rows and the content of the dragged column are automatically updated. This enables users to quickly reorder functions to find optimal sequence of function dependencies. The ability to move columns is illustrated in Figure 27 below.

![Figure 27: N-Squared View Function Reordering](image)

4.2.4 Check Project

The SSPARCy application contains functionality to help verify that interactions between functions are valid. To ensure proper interaction of the simulation, the system performs a series of integrity checks on the underlying code. Although the current suite
of checks does not guarantee proper interaction of simulation components, it is a useful tool in identifying disruptive errors and reviewing code interface assumptions.

The first check performed by the system involves function usage. SSPARCy checks that every function in a project is called by at least one other function. If a function is not called by another function, then no function utilizes its output, nor is it ever invoked during a simulation. This implies that the function is isolated from the rest of the simulation and is not essential. This check can help integrators pinpoint functions that are no longer needed as part of a simulation or that have been inadvertently severed from the flow of information.

The next check performed by the application concerns the proper use of simulation constants. For each function that uses the simulation wide constants, the application checks that each constant used in the code is defined in the simulation constants script. If the constant is not defined as a variable of the constants output structure of the constants script, then an error message is generated. This test ensures that a function does not attempt to use outdated or no longer existing constants.

Similarly, the application verifies correct usage of design variables. For each function that uses the design variables of the simulation, SSPARCy tests that the design variable script defines each design variable used in the code of the function. Therefore, each design variable employed by a function must be defined as a variable in the output structure of the design variable script of the simulation. This check helps to make sure that functions are not using invalid design variables.

Another test relates to the use of simulation constants. This test examines the constants script of the simulation. For each constant defined in the constants script, the
tool checks to see that another function uses that constant. If the constants script declares a constant that is not used by any function, then an error message is generated. This check identifies any unused or irrelevant constants.

Likewise, the script defining all design variables is also examined. For each design variable stated in the design variable script, the application tests to ensure that it is used by some other function in the simulation. This test discovers any unemployed design variables.

An additional check is performed to look at the validity of function interfaces. In this check, the tool looks at each input structure or field for each function. It then tests to make sure that the input field or structure is an output of a different function. This check provides a measure of the integrity of the interfaces of functions. It will catch any changes in the interface of a function that has not been propagated to functions with which it shares a dependency.

The last check conducted concerns the accurate use of variables of input structures. For each function, the application examines each input in turn. For each input, each variable is looked at. For each variable of a structure, the tool tests to see if the variable is defined as a variable of an output structure with the same name. If not, then an error flag is raised. This check addresses the proper usage of variables of output structures and makes sure that irrelevant variables are not used.

Errors are associated with the function responsible for them. When a user chooses to check a project using the ‘Check Project” menu and the ‘Show Errors” option, a list of the functions that have errors associated with them are displayed in a list format on the left hand side of the screen.
Figure 28: Project Errors View

Figure 28 shows a screenshot of the error checking functionality. When a function with errors is selected from the list on the left hand side of the screen, the errors associated with that function are displayed in the text area on the right hand side. This view provides a convenient way to organize errors and to allow the user to logically navigate through them.

4.2.5 Conclusion

The SSPARCy application offers a suite of features that provide users with a centralized source of information regarding a simulation. It presents the current state of a
simulation in order to enable a user to quickly access important information automatically. Additionally, it records the history of the various states of the entities of a simulation, allowing a designer to examine and analyze the evolution of a simulation over time. Furthermore, it performs automatic analysis of system integration integrity to alert an integrator of potential problems. This set of capabilities supplies a powerful set of tools to support the process of software integration and design rationale capture.
5 Technical Implementation

The SSPARCy application is capable of analyzing valid MATLAB code and extracting meaningful information from it. In order to extract information correctly, the application must be able to parse MATLAB code. This is accomplished by understanding the MATLAB syntax surrounding information of interest as well as being able to put this raw information into context. This context was specified by the SSPARC team and provides meaning to the parsed code for use in analysis and state capturing. Additionally, the tool makes numerous assumptions about valid MATLAB syntax and code conventions of the SSPARC team in order to process code properly.

For convenience of updating the application in the future, a constants file called Constants.java was added to the source file for the application. This file contains a list of constants that can be configured by the user to suite preferences or to adjust to changes in assumptions about simulations or MATLAB syntax. Figure 29 shows the constant fields defined in Constants.java.

```java
public class Constants {

    public final static String START_FUNC = "BTOS";

    //--------------file name constants-------------
    public final static String FILE_EXTENSION = "m";
    public final static String FILE_NAME_EXT_SEPARATOR = ".";
    public final static String DIR_SEPARATOR = "/";
    public final static String CONSTANTS_FILE_NAME = "Constants";
    public final static String DESIGN_FILE_NAME = "Design";

    //--------------code constants------------------
    public final static String CONSTANTS_CODE = "CONSTANTS";
    public final static String DESIGN_CODE = "DESIGN";
    public final static String COMMENT = "%";
    public final static String ASSIGN_VAL = "=";
    public final static String ARRAY_OPEN = "{";
    public final static String ARRAY_CLOSE = "}";
    public final static String ARRAY_NEXT = "...";

```
Figure 29: Application Constants

5.1 Information Extraction

5.1.1 Files

When a project is first created, the user provides the application with a directory name. This directory is assumed to contain all of the files that are part of the simulation to be modeled. If any files are part of the simulation, but are not in the specified directory, then the application will not include them in its representation. Furthermore, SSPARCy only seeks to model information generated by functions. To that extent, it assumes that all functions are specified in files ending in "m". Only files with the extension "m" will be analyzed. It should be noted that file names beginning with "#" are not analyzed, even if they end in "m". It is assumed that file names with the first character of "#" are files that have been deleted by the user, but not yet removed by the file system.
5.1.2 Parsing Out Strings

When the tool parses MATLAB code [10], it looks for the occurrence of certain key symbols or characters in the code body of a file. It assumes that the presence of these symbols and characters in certain contexts represents specific operations or information. One difficulty in automatically recognizing these key characters is when they happen to occur as part of a string (meaning a sequence of characters which are intended to be taken as literal text and not names for operations or information). When this takes place, the key characters are to be ignored and treated as literal text. Thus, when SSPARCy is looking for key characters, it must avoid paying attention to the content of strings.

This is accomplished by removing all strings from a line code when that line is analyzed. In MATLAB, a string is delimited by a single quote, ‘‘’. Thus, it is the goal of the application to remove all characters from ‘‘ to ‘‘, including the beginning and ending single quotes. Such an action would be straightforward if not for one perplexing difficulty in MATLAB. MATLAB also uses the single quote, ‘‘, symbol to represent the transpose operation. Thus, the code of “A’+B” consists of the transpose of array A plus the transpose of array B and does not contain a string “+B”. This creates a challenge for the application to recognize when the occurrence of single quotes represents a string delimiter or a transpose operation.

The following method (the term for procedure or function in the Java programming language [5]) implements the algorithm used by SSPARCy to remove all strings from a single line of MATLAB code. Note that the Java method ‘foo.indexOf("bar ")’ returns the index of the first occurrence of the first character of
public String parseStringsOut(String strLine) {
    String strCode = "";
    int iString = strLine.indexOf("'");
    //get rid of strings
    //first see if the line even contains a ' before a %
    StringTokenizer st;
    if((iString != -1) && ((iString < strLine.indexOf("%")) || (strLine.indexOf("%") == -1))) {
        //break line up by '
        st = new StringTokenizer(strLine, "'", true);
        //if only one ' then it can't contain a string
        String oldest = "";
        String oldestNo = "";
        String old = "";
        String oldNo = "";
        String current = "";
        String currentNo = "";
        while(st.countTokens() > 0) {
            currentNo = ((String) st.nextToken());
            current = currentNo.trim();
            if(oldest.equals("'") && current.equals("'")) {
                //old.startsWith(".+"), old.startsWith("-"), old.startsWith("**"), old.startsWith("^"),
                //old.startsWith("/"), old.startsWith("=="), old.startsWith("<")
                if(old.startsWith("=") || old.startsWith("+")) ||
                    old.endsWith("+") ||
                    old.endsWith("-")) ||
                    old.endsWith("\") ||
                    old.endsWith("<") ||
                    old.endsWith(">") ||
                    old.endsWith("=") ||
                    old.endsWith("%") ||
                    old.endsWith("+") ||
                    old.endsWith("-") ||
                    old.endsWith("\") ||
                    old.endsWith("<") ||
                    old.endsWith(">") ||
                    old.endsWith("=");
                strCode = strCode + oldestNo + oldNo;
                oldest = ";
                oldestNo = ";
                old = current;
                oldNo = currentNo;
            }
        }
    }
//include
strCode = strCode + oldestNo+oldNo+currentNo;
oldest = "";
oldestNo = "";
old = "";
oldNo = "";
}

continue;
} else {
    //string
    oldest = "";
    oldestNo = "";
    old = "";
    oldNo = "";
    current = "";
    currentNo = "";
}

if(st.countTokens() == 0) {
    strCode = strCode+oldestNo+oldNo+currentNo;
    continue;
}

strCode = strCode + oldestNo;

oldest = old;
oldestNo = oldNo;
old = current;
oldNo = currentNo;

}//closes while

} else {//closes if ' before %

return strLine;
}

return strCode;
}//closes method

Figure 30: Method to Remove Strings

The method shown in Figure 30 takes a complete line of code as input and returns the same line of code, only with all strings removed. The first test that the algorithm performs is to check to see if a "%" symbol occurs before the first " " in the line. The
"%" symbol represents a comment. All text on the same line following the "%" is a comment and is to be taken as literal text. If a "'" occurs after a "%" it cannot be the start of a string. Thus, if the first occurrence of a "'" takes place after a "%", then the current line of code contains no strings and the method returns the originally inputted line.

If the line does contain a "'" before any "%", then it could potentially contain strings. The next step the method takes is to partition the line of code by dividing it into chunks (tokens) of characters based on the presence of "'" symbols. A token consists of every character from the last "'" until the next one. Also, each "'" is considered to be a token as well. Thus, the first token is all of the characters from the start of the line until the first "'" unless a "'" symbol is the first character, in which case the "'" is the first token. The last token is all of the characters after the last "'" unless the last character is a "'"; in which case the last token is "'":

The algorithm utilizes the fact that all strings must occur between two "'". The basic function of the algorithm is to examine a token, one at a time, and always remember the last three looked at. If a token is older than the last three tokens examined and it has not been discarded, then it is added to the line of code that is to be returned. The only possible occurrence of a string is when three consecutive tokens are "'", "some text"", and "'". However, it is possible that such a sequence is not a string, but instead an operation involving transposes. For instance, "A'B" would result in three consecutive tokens of "'", "+B", and "'". Thus, the application must have a way to differentiate between the case of a transpose operation and a string.
SSPARCy distinguishes between a transpose operation and a string by exploiting the fact that the sequence of tokens "' ' 'some text', and "' ' 'can only be a transpose operation if the first character of the middle token is a key character. The valid MATLAB key characters that would imply an operation involving a transpose, instead of a string, would be all of the MATLAB operators or delimiters.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition, unary plus</td>
</tr>
<tr>
<td>-</td>
<td>Minus, unary minus</td>
</tr>
<tr>
<td>*</td>
<td>Matrix multiply</td>
</tr>
<tr>
<td>.*</td>
<td>Array multiply</td>
</tr>
<tr>
<td>^</td>
<td>Matrix power</td>
</tr>
<tr>
<td>.^</td>
<td>Array power</td>
</tr>
<tr>
<td>\</td>
<td>Left matrix divide</td>
</tr>
<tr>
<td>.\</td>
<td>Left array divide</td>
</tr>
<tr>
<td>/</td>
<td>Right matrix divide</td>
</tr>
<tr>
<td>./</td>
<td>Right array divide</td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
</tr>
<tr>
<td>~=</td>
<td>Not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>Logical NOT</td>
</tr>
<tr>
<td>`'</td>
<td>Quote, complex conjugate transpose</td>
</tr>
</tbody>
</table>

82
Figure 31: MATLAB Operators and Delimiters

| .   | Transpose |
|     |           |
| ( ) | Parentheses |
| { } | Braces |
| [ ] | Brackets |
| ,   | Comma |
| =   | Assignment |

Figure 31 lists all of the valid MATLAB operators and delimiters. If the middle token starts with (excluding white space) any of the operators listed in Figure 31, then the preceding token (" ") is a transpose operation and should be added to the line of code to be returned. Additionally, the middle token should be added since it is not preceded by a "" which represents the start of a string.

Given the information so far, it is ambiguous whether or not the last token, " ", is a transpose operation or the start of a string. This ambiguity exists because of the validity of having an operation of the form "A'+'this is a string" where "A" is an array and "A'" represents the transpose of the array "A" and "this is a string" is a string of text. However, this ambiguity can be resolved by noting that if the last character of the middle token (excluding white space), which is also the first character before the last token " ", is an operator or delimiter, then the last token " " cannot be a transpose operator because it is invalid in MATLAB to have a variable name containing those symbols. If none of these characters immediately precede the last token, then the last token must be a transpose operator and should be included in the line of code returned. Otherwise, it is the start of a string and should remain in memory to be evaluated with its
subsequent tokens. Note that once a token has been added to the line of code to be returned, it is removed from memory.

After each token has been examined, if it is not removed from memory, it is stored in the second memory slot ('bld'). The previous token in the second memory slot is moved to the first slot ('bldest'). The token already in the first memory slot is appended to the line of code to be returned. If the last token is reached, then all tokens in memory are added to the line of code.

Thus, the algorithm implemented in Figure 30 divides a line of code into tokens by the occurrence of the ‘‘” symbol. It looks for three consecutive tokens of “”, “text”, and “”. If a token is older than the three most recent tokens examined, then it is appended to the line of code, assuming the token has not been discarded. Tokens are only discarded if they are part of a string, which is identified by the occurrence of the three consecutive tokens “”, “text”, and “” where the first character of the middle token is not a MATLAB operator or delimiter as listed in Figure 31.

This algorithm has the limitation that it will function improperly if any strings begin or end with a MATLAB operator or delimiter. If this occurs, it will fail to remove the offending strings from the line of code. This can potentially result in improper character recognition in other extraction operations of the application if characters being looked for occur in the offending strings. However, as long as this constraint is adhered to, the algorithm will correctly remove all strings from a line of MATLAB code.
5.1.3 Parse Out Comments

The existence of comments in the body of code presents a similar problem to that of the presence of strings. All commented text is supposed to be regarded as literal text. However, the SSPARCy application examines code for the occurrence of specific character sequences. If these sequences take place in commented text, the application should disregard them. This requires that SSPARCy know when it is examining commented code. The easiest solution to accomplish this is to remove commented code when performing analysis. Figure 32 shows the implementation of a straightforward procedure for removing comments from a single line of code.

```java
private String parseCommentsOut (String strLine) {
    String strCode;
    if (strLine.indexOf (Constants.COMMENT) != -1) {
        int iEndInd = strLine.indexOf (Constants.COMMENT);
        strCode = strLine.substring(0, iEndInd);
    } else {
        strCode = strLine;
    }
    return strCode;
}
```

Figure 32: Method to Remove Comments

The method shown in Figure 32 takes a line of code as input and returns the same line of code with all comments removed. Note that this method is called after the method for removing strings from a line of code has been called. This is necessary so that the comment removal method does not incorrectly identify the start of a comment when it occurs in a literal string.
The comment removal method begins by looking for the index of the first occurrence of the comment symbol ‘%’. If this symbol does not exist in the input line of code, then the original line of code is returned, unmodified. However, if the line of code does contain the ‘%’, then all characters following the symbol are commented text. The method removes all characters after the ‘%’ symbol, as well as the symbol itself, from the line of code. The remaining code is returned.

Since the ‘%’ symbol is not an operator other than the comment symbol, there is no need to figure out the context of its usage. All the application must be sure of is that the occurrence of the symbol it identifies does not occur in a string. Calling the method to remove strings from a line of code first and passing the resulting line to the method for removing comments achieves this. This ensures that all ‘%’ identified are not part of string.

For the remainder of this section 5, when a reference is made to a line of code, it will be assumed that this line does not include any strings or comments unless otherwise specified.

5.1.4 Functions

The application first retrieves the name of all files ending in ‘.m’ from the user specified directory. It is assumed that each file corresponds to a function. Thus, the tool constructs a representation for each function/file.

Extraction of information relating to a file begins by looking for the first occurrence of a line of code with the word ‘function’ in it. The presence of the word
'function' indicates the declaration of a function. When this line is discovered, the application first checks to see if the function declaration spans more than one line of code. A multi-line declaration is indicated by the existence of a "..." array next symbol on the first line of the declaration. If this occurs, it indicates that either the inputs or outputs are listed on more than one line. Thus, all successive lines of code are appended to the first line until a "}" (indicating the end of multiple outputs) or "}" (indicating the end of multiple inputs) is reached on a successive line other than the first. Note that the "..." symbol is also removed. The implementation for this practice is shown in Figure 33.

```java
//see if this function declaration contains a continuation
//symbol "...
if(strCode.indexOf(Constants.ARRAY_NEXT) != -1) {
    //if continuation, then must be of outputs or inputs
    //if so, read in successive lines until you reach a
    //"}" or ")"
    //append to the first line, minus the "..."
    strCode =
    strCode.substring(strCode.indexOf(Constants.ARRAY_NEXT) + Constants.ARRAY_NEXT.length());

    while(true) {
        strLine = brFunction.readLine();
        String strTmp;
        if(strLine.indexOf(Constants.COMMENT) == -1) {
            strTmp = strLine;
        } else {
            int iEndInd = strLine.indexOf(Constants.COMMENT);
            strTmp = strLine.substring(0, iEndInd).trim();
        }
        strCode = strCode + strTmp;
        if((strTmp.indexOf(Constants.ARRAY_CLOSE) != -1) || (strTmp.indexOf(Constants.FUNC_CLOSE) != -1)) {
            break;
        }
    } //closes while
}
```
Once a multi-line declaration has been modified to fit on a single line, it is assumed that all function declarations (multi-line and single) take one of the following forms.

```
function [output1, output2] = name(input1, input2)
function [output1, output2] = name
function output1 = name
function output1 = name(input1, input2)
```

The first task of the application is to extract the name of the function as declared in the function declaration line. The name of the function is vital because it is the key character sequence used by other functions to invoke the corresponding function. Given the enumeration of possible declarations listed in Figure 34, the name of the function is the sequence of characters (excluding white space) immediately following "=" and until the first occurrence of "(" or the end of the line. Thus, the method extracting the function names checks for the presence of the "=" and "(" symbols. If both are present, it extracts the characters immediately after the "=" until the "(". If only the "=" is present, then it takes the all of the characters on the line succeeding the "=" symbol. This procedure assumes that the first occurrence of the "(" symbol never precedes the first occurrence of the "=" symbol.

Not all functions have an explicit function declaration in their code bodies. Such functions are usually the start function and are responsible for not returning any output to another function, but instead interfacing with the user. For such functions where the
application fails to discover a function declaration line, it simply takes the name of the file, excluding the “.m” extension, as the name of the function.

It should be noted that the application only searches for the first occurrence of the function declaration line. Sub-functions are declared using the same format as functions in MATLAB. Their declaration lines follow somewhere in the same code file as the function declaration. Thus, the tool does not recognize sub-functions. This is a reasonable practice since the scope and visibility of sub-functions is limited to the parent function and the goal is to consider interaction between parent functions.

It should be noted that the application assumes that no two functions of the same project will have the same name, even if they are declared in separate files.

5.1.5 Function Inputs

Once a function declaration line has been identified and concatenated into a single line of code, the process of extracting function inputs is straightforward. As indicated in Figure 34, all inputs are declared between the ‘(‘and ‘)’ symbols of the function declaration line. Thus, all characters between the ‘(‘ and ‘)’ symbols are extracted, divided by the input separator ‘,’; and each resulting division is an input that is associated with the given function. If the ‘(‘ and ‘)’ symbols are not present, then this indicates that the given function is a script and has no inputs. If so, an empty set of inputs is associated with the script.
5.1.6 Function Outputs

The method for extracting outputs from a function declaration line is similar to that of extracting inputs. Figure 34 demonstrates how all outputs of a function are delimited by the ‘[“and ‘]” symbols; if it is a single output, it is enclosed by no symbol. If the function declaration line contains a ‘[“and ‘]” symbol, then all enclosed characters are extracted, divided by the output separator “;”, and the resulting divisions are outputs associated with the given function. If the ‘[“and ‘]” symbols are not present, then the character sequence following ‘function’ up until “;” are taken as the output and associated with the given function.

5.1.7 Function Calls

Calls made to other functions by a given function are recorded by the application. This information is vital because it reveals the direct dependency between functions. After all of the names of the functions of a project have been determined by examining the function declaration lines of each file, it is then possible to scan each function code body to look for calls to another function. A call to another function takes place if the name of the function appears alone in the code. Figure 35 shows valid MATLAB function calls.

```matlab
name    // for script calls
name(input1)   // for function/module calls
```

Figure 35: **Valid MATLAB Function Calls**
Thus, once the application has discovered the name of a function, it must begin its analysis to determine if it is a valid function call. The fact that function names can be used as subsequence of character sequences in MATLAB necessitates this analysis. For instance, if a function named ‘foo” exists, a valid variable name is ‘Barfoo”. The fact that a line of code contains the character sequence corresponding to the name of a function does not necessarily imply that the line is making a call to the function. Instead, the context of the name usage must be examined.

Figure 36 shows the Java implementation of the method used to determine if a valid function call occurs on a given line of code.

```java
private Call setupCall(Enumeration eFunctions, String strLine) {
    //go through all of the possible function/script names
    while(eFunctions.hasMoreElements()) {
        String strFunctionName = (String) eFunctions.nextElement();
        //strFunctionName = strFunctionName+Constants.FUNC_OPEN;

        strLine = strLine.trim();
        //see if this line contains "name"
        int iIndex = strLine.indexOf(strFunctionName);
        if(iIndex != -1) {
            //if so, and the name is not the beginning of the line
            if(iIndex != 0) {
                //if the function name is not the first char of the line
                //then get the preceding char
                char c = strLine.charAt(iIndex-1);
                //check to make sure an op,
                //done to ensure name is not a substring
                if(!(c == '(' || c == '[' || c == ' ' || c == '+' || c == '-' || c == '*' || c == ')' || c == '\' || c == '/' || c == '=' || c == '<' || c == '>' || c == '&' || c == '|' || c == '~' || c == ',') ) {
                    continue;
                }
            }
        }
    }
}
```
The method shown in Figure 36 takes a single line of code not containing any strings or comments as input. In addition, it takes in a list of names of all functions as input. It outputs a Call object, which is how the system represents a call to another function, if a function call occurs. Otherwise the method outputs a null object.

The method loops through each function name in the input list and sees if it occurs in the line of code. If so, then this is potentially a function/module/script call as shown in Figure 35. It is possible that the function being check for in the current line is also the ending character sequence of another function name. For instance, assume we are checking to see if the current line contains a call to a function named "foo". However, there also exists a function called "barfoo". Thus, if the present line contains a
call to the function ‘barfoo”, then the application will successfully find the character sequence ‘foo“ in the line. However, this sequence should not be credited as a call to ‘foo”, but to ‘barfoo” instead. To ensure that ‘foo” is not a sub-sequence of any name, the application must be certain that the character immediately preceding ‘foo” is blank space or any of the MATLAB operators listed in Figure 31. If any of these characters is the character just before the function name, then it is potentially a valid call. To be certain, the character immediately following the function name must be checked as well. If it is a blank space or any of the characters in Figure 31, then this is an occurrence of a valid function/module/script call. The application notes this function call and returns.

If any of the above validity checks fail, the method continues on to the next function name. If no function names are detected for a line of code, then null is returned.

5.1.8 Input Structure Variables

SSPARCy tracks the fields used of any input structure passed to a function. For instance, if function ‘foo” has the input structure ‘BAR”, SSPARCy will attempt to find all variables of the input structure ‘BAR” that are used in the code body of ‘foo”.

The task of extracting variable names from code is similar to that of finding function calls in the code (see previous section). The method for finding input structure variables examines each line of a function file in turn. First, it removes all strings and comments using the respective methods shown in Figures 30 and 32. Next, it checks to ensure that the line is not a function declaration. If it is, then it proceeds to the next line of the file. Then it removes all code for a line that comes before an “=” symbol. An “=”
symbol is an assignment operator and therefore can only be assigning local or output variables. Thus, all code before, and including the “=” symbol, can be ignored. The Java code for removing all code not after the “=” is shown in Figure 37.

```java
//get the portion of the code following an "="
if(strCode.indexOf(Constants.ASSIGN_VAL) != -1) {
    int iAssign = strCode.indexOf(Constants.ASSIGN_VAL);
    int assign[] = new int[4];

    //see if the "=" is part of one of the following
    assign[0] = strCode.indexOf("==");
    assign[1] = strCode.indexOf(">=");
    assign[2] = strCode.indexOf("<=");
    assign[3] = strCode.indexOf("~=");

    //find the first occurrence of the above
    int lowest = strCode.length();
    for(int i=0;i<assign.length;i++) {
        if(assign[i] < lowest && assign[i] != -1) {
            lowest = assign[i];
        }
    }

    //if the index of "=" is less than any of the above than
    //the line must have an assignment in it.
    if((iAssign < lowest) && (iAssign != -1)) {
        //get all code after the assignment
        strCode = strCode.substring(iAssign+1);
    }
    //otherwise, no assignment
}
```

**Figure 37: Algorithm to Extract Text After Assignment**

In Figure 37, “strCode” represents the current line of code (void of comments and strings). The difficulty in extracting the code following an assignment lies in correctly identifying an assignment. The simple presence of the “=” symbol does not imply an assignment. The operators “==”, “>=”, “<=”, and “~=” all contain the “=” symbol but are not assignments. An assignment exists only if the “=” symbol exists alone. If it exists alone on the same line with one of the operators using the “=” symbol, then the
assignment operator will precede any of them. Therefore, the algorithm in Figure 37 finds the index of all operators involving the "=" symbol. If the assignment symbol has the lowest positive index, then it is a stand-alone "=". If its index is the same or greater than any of the operator indices, then it is not an assignment symbol, but instead part of another operator. If the line does indeed contain an assignment symbol, then all code after the symbol is retained. Otherwise, the whole line is retained.

Once the application has extracted the portion of a line of code after the assignment symbol, it next looks for the presence of any inputs. More precisely, it loops through each input to the function and checks if it occurs followed by a "." symbol in the current line. If it finds an occurrence, then it checks to ensure that the name of the input structure is not the ending character sequence of another input structure. It accomplishes this by checking that the character immediately preceding the given input structure name is a valid MATLAB operator as listed in Figure 31. If so, then the character sequence we have identified is a use of a variable of the given input structure.

This implies that the name immediately following the "." is the name of the input structure variable. Again the tool encounters the difficulty of determining the end of a character sequence. This task is achieved by looking for the first valid MATLAB operation or ";" occurring after the "." symbol. It is assumed that the variable name is a set of all characters from the "." to the first tested symbol. The variable name is then extracted and associated with the containing input structure.

An interesting problem arises when multiple uses of input structure variables occur on the same line. For instance, assume a function has two input structures: "FOO" and "BAR". An example of a challenging line to analyze is "FOO.var1 + BAR.var1 +
This example contains three variables that should be extracted and recorded. Figure 38 shows an algorithm for finding the names of all variables of all input structures for a particular line of code.

```java
//loop through the names of the inputs
while(e.hasMoreElements()) {  
    String strThis = strCode;
    strThis = strThis.trim();

    //get input name
    String strName = (String) e.nextElement();
    String strTest = strName + Constants.STRUCT_FIELD_SEP;

    //see if this line contains an input structure's name + "." 
    if(strThis.indexOf(strTest) != -1) {
        //while there are still characters to examine in this 
        //string...
        for(int iInd=0;strThis.length() > 0; ) {
            //makes sure input structure's name is not a
            //valid occurrence of name
            if(strThis.indexOf(strTest) == 0) {
                //implies not a valid use
                //examine the rest of the line
                strThis = strThis.substring(strThis.indexOf(strTest)+strTest.length());
                continue;
            }
        }
    }
}
```
//thus a structure
//then get the name of the field.
//start with the point right after the .
iBegInd = strThis.indexOf(strName)+strName.length()+1;
iEndInd = 0;

//guess all possible endings for the name of the
field
int endings[] = new int[30];
endings[0] = strThis.indexOf(";", iBegInd);
endings[1] = strThis.indexOf("+", iBegInd);
endings[2] = strThis.indexOf("-", iBegInd);
endings[3] = strThis.indexOf("*", iBegInd);
endings[4] = strThis.indexOf("^", iBegInd);
endings[5] = strThis.indexOf("\\", iBegInd);
endings[6] = strThis.indexOf("\", iBegInd);
endings[7] = strThis.indexOf("=", iBegInd);
endings[8] = strThis.indexOf("->", iBegInd);
endings[9] = strThis.indexOf("<", iBegInd);
endings[10] = strThis.indexOf("">", iBegInd);
endings[11] = strThis.indexOf("&", iBegInd);
endings[12] = strThis.indexOf("|", iBegInd);
endings[13] = strThis.indexOf(":", iBegInd);
endings[14] = strThis.indexOf("), iBegInd);
endings[15] = strThis.indexOf("{", iBegInd);
endings[16] = strThis.indexOf("}", iBegInd);
endings[17] = strThis.indexOf("\], iBegInd);
endings[18] = strThis.indexOf("\]", iBegInd);
endings[19] = strThis.indexOf("\{", iBegInd);
endings[20] = strThis.indexOf("\}", iBegInd);
endings[21] = strThis.indexOf("\[", iBegInd);
endings[22] = strThis.indexOf("\], iBegInd);
endings[23] = strThis.indexOf("\{", iBegInd);
endings[24] = strThis.indexOf("\}", iBegInd);
endings[25] = strThis.indexOf("\]\), iBegInd);
endings[26] = strThis.indexOf("\[^", iBegInd);
endings[27] = strThis.indexOf("\^\", iBegInd);
endings[28] = strThis.indexOf("\\\", iBegInd);
endings[29] = strThis.indexOf("\", iBegInd);

//see which is the ending for the name by
choosing smallest

//index greater than the beginning index.
for(int i = 0;i<endings.length;i++) {
    if((endings[i]>iBegInd)) {
        if(iEndInd == 0) {
            iEndInd = endings[i];
        } else if(endings[i]<iEndInd) {
            iEndInd = endings[i];
        }
    } //closes if
} //closes for

//if none other the above terminations, then
//must be a line with no termination
if(iBegInd > iEndInd) {

//print the name of the field

}
iEndInd = strThis.length();
}

//get the name
String strField = strThis.substring(iBegInd, iEndInd).trim();

Figure 38: **Algorithm for Discovery of Input Structure Variable Names**

The above algorithm begins by looping through each input to a function. For each input, it finds all of the occurrences of the structure name in the given line. To accomplish this, it finds the first occurrence and determines the variable name as described previously in this section. Once the variable is recorded, the application removes all characters before and including the newly discovered variable name. The algorithm then examines again the new abbreviated line for another occurrence of the current input structure. In this fashion, the algorithm repeatedly shortens a line of code as it discovers input structure variables until the line contains no more characters or no additional uses of the current input structure. When this analysis has been completed for one input structure, it is begun again with the next input structure name and the entire original line of code. By repeating this process for each input structure name and then for each line of code, the application is able to extract the occurrence of multiple uses of input structure variables in the same line of code.

With the ability to determine the valid beginning and end points of an input structure variable use, and the capability to identify multiple valid uses in the same line of code, the application can extract and capture all variables of every input structure that is used in the code body of a function.
5.1.9 Output Structure Variable

The SSPARCy tool records the variables of output structures defined by each function. To extract the variable names, the algorithm exploits the fact that all output structure variable names precede the occurrence of the first assignment symbol in a line of code. The presence and position of the first assignment symbol is determined in the same fashion as described in section 5.1.8 Input Structure Variable. If an assignment is present on a given line, then all of the code following and including the assignment symbol are removed from the line.

The line is then examined to see if it contains the name of an output structure. If it does, then the tool checks to ensure that the name is not the sub-sequence of some other name. This is determined by testing that the structure name is the first character on the line or is preceded by a blank space. In addition, the name must be immediately followed by a ".". If the occurrence of the output structure name is valid, then the variable name is assumed to be all of the characters (minus white space) from the character after the "." to the end of the line (which is equivalent to all the characters up until the assignment symbol in the original string). Note that the algorithm described here assumes that assigning values to an output structure variable follows the subsequent form:

```plaintext
FOO.var1 = some value;
```

**Figure 39: Valid Assignment of a Value to an Output Structure Variable**

This form does not permit the presence of multiple structures before the assignment symbol. Because of this restriction and the constraint that operations cannot be
performed to the left of an assignment symbol, the process of determining output structure variables is straightforward in comparison to discovering other information.

5.1.10 Constants

SSPACy also captures the constants and their associated values for a simulation. It is assumed that all constants are defined in a file named ‘constants.m’. The name of the file is not case sensitive, however, it must solely consist of the name ‘constants.m’. Note that altering the constant CONSTANTS_FILE_NAME in the Constants.java file can change the name for the constants file.

Once the constants file has been opened, each line is examined after all comments and strings have been parsed out. If the line contains a function declaration, it is skipped. Otherwise, it is checked to see if it contains an assignment operator following the name of the output structure of the constants script. The name of the output structure of the constants script is also specified in the Constants.java file by the constant field CONSTANTS_CODE. Also, the algorithm described in section 5.1.8 Input Structure Variables checks for the presence of the assignment symbol. If so, then the name of the constant is extracted in the same fashion that an output structure variable name is. The characters from the ‘:’ symbol to the assignment symbol (excluding the assignment symbol) are taken (excluding white space) to be the constant name.

An interesting challenge arises when attempting to extract the value associated with a constant. The simplest case is when the value of a constant consists of all the code
on the given line following the assignment symbol to the end of the line. However, it is possible for the value to be declared on multiple, consecutive lines.

```java
CONSTANTS.con1 = value;
CONSTANTS.con2 = [value];
CONSTANTS.con3 = ...;
CONSTANTS.con4 = [

Figure 40: Possible Single Line Value Declarations
```

As Figure 40 shows, there are two possible value declarations that span more than a single line. Thus, if the algorithm detects a “...” or a “[” without a corresponding “]”, then it recognizes the value as a multi-line declaration. When this occurs, the algorithm reads successive lines until it finds one containing a “]”, which indicates the termination of a multi-line declaration. All of the lines of the multi-line declaration are then associated with the constant as its value. The algorithm for extracting multi-line value declarations is shown in Figure 41.

```java
//now get the value
//first, check to see if the value begins with a [ or ...
//which indicates a multi-line declaration of the value.
String strValFull = strCode;
String strVal = "";

if((strValFull.indexOf( Constants.ARRAY_OPEN) == -1) && (strValFull.indexOf( Constants.ARRAY_NEXT) == -1)) {
    strVal = strValFull.substring(0,
strValFull.indexOf( Constants.END_OF_LINE)).trim();
} else if((strValFull.indexOf( Constants.ARRAY_CLOSE) != -1) && (strValFull.indexOf( Constants.ARRAY_OPEN) != -1)) {
    int iEndIndex =
strValFull.indexOf( Constants.ARRAY_CLOSE);
    int iBegIndex =
strValFull.indexOf( Constants.ARRAY_OPEN);
    strVal = strValFull.substring(iBegIndex, iEndIndex+1);
} else {
    //if it is a multi-line declaration, then remember all
    //values and keep looping through new lines until you
reach
    //a "]" which indicates the closing of the value.
    while(true) {
```
// first get the non-commented portion
if (strValFull.indexOf(Constants.COMMENT) != -1) {
    int iEInd = strValFull.indexOf(Constants.COMMENT);
    strValFull = strValFull.substring(0, iEInd);
}

// check to see if "...
if (strValFull.indexOf(Constants.ARRAY_NEXT) != -1) {
    // if so, skip and read next line
    strValFull = brFunction.readLine();
} else if (strValFull.indexOf(Constants.ARRAY_OPEN) != -1) {
    // see if ",
    strVal = strValFull;
    strValFull = brFunction.readLine();
} else if (strValFull.indexOf(Constants.ARRAY_CLOSE) != -1) {
    // see if "]
    // if so, read everything including the "]" and appended it as a newline to the existing answer
    int iEndIndex = strValFull.indexOf(Constants.ARRAY_CLOSE);
    strVal += \n"\n"+strValFull.substring(0, iEndIndex+1);
    // stop reading this value
    break;
} else {
    // otherwise, add the current line as a new line
    // and read the next line
    strVal += \n"\n"+strValFull;
    strValFull = brFunction.readLine();
}
// closes while
}// closes else for multi-line declarations

**Figure 41:**  **Algorithm to Extract Multi-line Value Declarations**

With the ability to extract single and multi-line values for each constant, the application is successfully able to retrieve all constant names and corresponding values without implying any constraints on the coder.
5.1.11 Design Variables

The SSPARCy system also tracks the names and values of simulation design variables. Just as with the extraction of constants information, the application assumes that the file declaring the design variables and values has a case insensitive name as specified by the DESIGN_FILE_NAME field in the Constants.java file of the SSPARCy source code. By default, the field is set to ‘Design.m”, however, any variation of lower or upper case characters will also work. Additionally, the assumed name of the output structure of the design variable script is specified in the DESIGN_CODE field in the Constants.java file. This is presently set to "DESIGN” and is case sensitive.

The algorithm for extracting design variable names and values is exactly the same as that for discovering constants, only adjusted to search in the Design.m file and for the output structure of “DESIGN”. Thus, the application supports the successful extraction of all design variable names and values, including multi-line values.

5.2 Check Project

The check project functionality performs seven integrity tests to ensure the validity of simulation integration. One assumption underlying several of the tests conducted by the application is that the same name is used for reference to the same input/output structure/field in all functions. Thus, if the output of one function is ‘FOO’ and another function uses this structure as an input, then it also calls it ‘FOO’. Equally important is that functions do not have name collisions for their inputs and outputs when
they do not share the same input/output. For example, a function should not have an
input ‘FOO’ if another function has an output ‘FOO’ and the function is not referring to
the same output when using the name as an input. MATLAB does not enforce these
naming conventions. Instead, simulation designers should follow these naming
guidelines.

The first test checks that every function is called by at least one other function,
except for one function that is designated as the start function. The start function is
invoked by the user or called by a function outside of the simulation. The name of the
start function is specified in the Constants.java file by the field START_FUNC. For all
other functions, each one is examined in turn. For each function, all other functions are
checked to see if they contain the current function in its list of calls to other functions. If
no other function contains the current function in its list of called functions, then an error
is generated stating, ‘function <name> not called by another function’.

The next test ensures that the constants script defines any constants used by a
function. Looping through each function performs this test. For each function that has
‘CONSTANTS’ as an input, the list of input structure variables for the structure
‘CONSTANTS’ is examined. Each input structure variable is checked to make sure that
its name is in the list of constants defined by the constants script. If not, the error
message ‘constant <name> does not exist in constants file’ is created. Note that this
check assumes the name of the output structure of the constants script and the input
structure to a function that contains the constants has the same name, which is defined in
the Constants.java file by the field CONSTANTS_CODE. By default this value is
‘CONSTANTS’.
The test for the proper use of design variables is similar to that for constants. For each function with an input of 'DESIGN', the input structure fields associated with 'DESIGN' are checked. Each variable is checked to ensure that it is present in the list of design variables outputted by the design variable script. The actual name of the design variable output structure is defined in the DESIGN_CODE field of the Constants.java file. If the design variable is not defined by the design variable script, then the error message "design variable <name> does not exist in design variable file" is created.

Reflexively, the application also tests to be certain that all constants defined by the constants script are used by some function. For each constant defined as output by the constants script, the list of input structure variables for the structure specified by CONSTANTS_CODE is retrieved from each function with CONSTANTS_CODE as an input. If none of these lists contain the constant, the error message "constant <name> is not used in any function" is generated.

Also, the tool checks that all design variables created by the design variables script are used by some function. For each design variable specified as output by the design variables script, the list of input structure variables for the structure corresponding to DESIGN_CODE is examined for each function with DESIGN_CODE as an input. If the design variable is not contained in any of the lists, then the error message "design variable <name> is not used in any function" is produced.

Another test performed checks that all inputs to a function are the outputs of some other function. For each function, the list of inputs is retrieved. For each input, the output list of every other function is examined. If none of them contain the input, then the error message "Input <name> is not the output of any function" is created.
The last verification conducted is to confirm that all input structure variables used are defined in the output structure of another function. This problem is equivalent to checking that all input structure variables are the output structure variables of another function. For each input structure variable, every other function is looked at. If the function has an output structure with the same name as the input structure, then the output structure variable list for that structure is retrieved. If none of the discovered output structure variable lists contain the input structure variable, then the error statement "Input field <nameVariable> of structure <nameStructure> is not defined in any output structure". This test will handle the case when multiple functions output the same output structure, but have different variables for that structure.
6 Comparison of SSPARCy to Other tools

The SSPARCy application automatically extracts information from MATLAB source code, represents and stores this information, performs analysis of the integrity of software integration, and allows the ability to capture the history and rationale associated with elements of a project. No existing tool aggregates all of these features, giving SSPARCy an advantage over other single faceted applications. However, there are a variety of tools that do offer a subset of the features available in SSPARCy. These tools offer functionality in one of three dimensions: Information Extraction, Software Integration, or Design Rationale Capture.

6.1 Information Extraction

A variety of tools exist for extracting information from source code files. Since programming languages vary in syntax and semantics, no universal tool for identifying elements in every programming language exists. Instead, tools are designed to follow rules specific to a single programming language in order to parse source code.

SSPARCy is one such tool. Its ability to process source code is specific to MATLAB syntax, just like MATLAB compilers and debuggers. However, SSPARCy possesses several advantages over other information extraction tools. SSPARCy is multi-faceted. It permits users to view information regarding the current state of the system, just like DDD and debuggers. Even in this respect, SSPARCy is superior to its peer information tools because of its ability to record previous states of a program and its
ability to allow users to view this history in a meaningful way. Furthermore, its capacity
to perform integration integrity checking and rationale capture makes it a much more
complete integration application. Other information extraction tools are intended to help
a single designer understand how a single piece of code operates. These tools are capable
of displaying the runtime value of variables and can step through the execution of a
program. SSPARCy is not intended to run a simulation of a program. Instead, it is an
integration support tool. Thus, it offers a suite of features that support the integration
process, not the debugging process. In this capacity, it is superior to tools such as
debuggers, compilers and DDD.

WAVE is an algorithm to learn information extraction rules. Since it is simply an
algorithm, it is not a suitable substitute for the broad functionality available in SSPARCy.
However, the incorporation of the WAVE algorithm into the SSPARCy application tool
for the purpose of empowering the tool to learn extraction rules over time could be a
significant potential enhancement. An implementation of the WAVE algorithm in
SSPARCy would augment its flexibility and ability to adapt to changes in MATLAB
syntax. Furthermore, it could potentially enable the application to analyze other
programming language by allowing SSPARCy to learn the information extraction rules
for a new language over time.

6.2 Software Integration

The present field of software integration systems requires the formal specification
of correctness requirements for a program. This specification must be implemented in
some formal notational language. This process requires significant overhead on behalf of
the program integrator to rigorously specify the properties of correct behavior in formal language. Such practices are necessary in order to use tools such as SPIN, COOL:Spex, Catalysis, the Boeing applications and the package of MSC, POGA, and TEMPLE. Systems that require formal specification of correctness requirements allow the integrator to fully specify what the definition of proper aggregate program behavior is on a case-by-case basis. Alternatively, SSPARCy does not seek to understand the context of correct behavior. Instead, it checks the basic integration integrity of a simulation. The series of tests it performs are generic in nature and are necessary for the proper functioning of any simulation. This limits SSPARCy from performing context specific integration tests. However, SSPARCy requires no additional overhead on the part of a simulation integrator to specify correctness requirements. Furthermore, SSPARCy conducts integration tests on code details that are abstracted away by SPIN, COOL:Spex, Catalysis, the tools from Boeing and MSC, POGA, and TEMPLE. This gives it greater capability over its peers to verify successful code integration independent of semantics. The potential to add the functionality to conveniently specify additional formal requirements is a possibly useful addition to the SSPARCy system.

The existing SSPARC tools represent the closest peer in terms of overall system features. The SSPARCy application can be best understood as the next generation of the SSPARC tools. SSPARCy offers nearly all of the functionality available in the SSPARC tools, but with several key additions. One significant enhancement that SSPARCy offers over the existing tools is that it automatically extracts information from the source code files. With the previous tools the simulation designers would have to manually input all information into the Excel spreadsheets. Thus, SSPARCy eliminates substantial
overhead, removes human error, and possesses a greater ability to maintain the consistency between the application state and that of the source code by performing accurate and automatic updates.

Another important improvement provided by SSPARCy is its ability to capture the history of the state of a simulation. The present SSPARC tools keep no record of the previous state of simulations. SSPARCy not only maintains a history of a project, but also displays the information in a visually distinct manner. Furthermore, by allowing rationale to be specified for each element of a simulation and recording this information with the given state, SSPARCy provides a previously non-existent design rationale capture capability. Users can now examine previous program states and view the rationale associated with key decisions. This provides a logical sense of the evolution of a simulation, a feature not available with the old SSPARC tools.

SSPARCy is a functional replacement for some of the existing SSPARC tools. The only feature presently not offered by SSPARCy that was used in the previous SSPARC tools is the information flow diagram shown in Figure 6. The system currently tracks the information necessary to produce such a diagram, however, the graphical implementation has not yet been specified and performed. This is a planned near term enhancement for SSPARCy. Overall though, the SSPARCy tool represents an evolution of the SSPARC tools by providing automation, program history and design rationale capture functionality.
6.3 **Design Rationale Capture**

The current set of design rationale capture tools spans the spectrum from dealing with fully unstructured rationale to completely modeled rationale. Meeting minutes represent an unstructured, time delineated capture. QuestMap and DRAMA represent the next step, providing basic structural elements and enabling the user to devise a useful structure. At the other end of the spectrum from meeting minutes is DRIM. DRIM is a completely specified model for the rationale underlying the design process. As a design rationale capture tool, SSPARCy lies somewhere on the spectrum between QuestMap and DRIM. SSPARCy creates a simple structure for design rationale by associating rationale with each simulation entity. Additionally, SSPARCy captures this rationale over time. Thus, rationale can evolve at any level from project to variable over time. This is a minimal logical structure. It provides flexibility for the user to specify rationale in whatever manner and at whatever level is most beneficial. It does not impose the rigid conceptual structure that DRIM proposes. Therefore, SSPARCy is a compromise in terms of design rationale capture between inflexible structure and amorphous disorder. Furthermore, the basic structure it provides is the most appropriate one for the domain-specific design process it endeavors to capture.

6.4 **Comparison Summary**

The following table provides a summary of the comparison of SSPARCy to all of the surveyed tools. The first column "Automated" refers to the ability of the application
to gather information automatically. ‘MATLAB’ concerns the capacity of the system to process and extract information from MATLAB code, which is the implementation language used by the SSPARC team. ‘View’ addresses the functionality to view gathered information in a meaningful way. ‘History’ speaks to the ability of a tool to record previous state information. ‘Check Integrity’ summarizes the facility to perform integration integrity tests. Finally, ‘DRC’ is reviewing the presence of a design rationale capture feature.

<table>
<thead>
<tr>
<th>Name</th>
<th>Automated</th>
<th>MATLAB</th>
<th>View</th>
<th>History</th>
<th>Check Integrity</th>
<th>DRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSPARCy</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Information Extraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MATLAB debugger [13]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MATLAB compiler [13]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DDD [26]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>WAVE [4]</td>
<td>Yes</td>
<td>Possibly</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Software Integration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPIN [6]</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MSC, POGA and TEMPLE [1]</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>COOL:Spex [29]</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Catalysis [28]</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Boeing Tools [27]</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SSPARC Tools [14] [15]</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Design Rationale Capture</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minutes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>QuestMap [11]</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DRAMA [3]</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DRIM [8]</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 42: **Comparison of SSPARCy to Other Tools**
7 Future Research

The SSPARCy tool has undergone tremendous evolution from originally being a potential value-add application of information technology, to a proof of concept demonstration, and then to a useable tool. The next phase for SSPARCy will be to enhance its present graphical functionality to provide all of the features and convenience needed to support the integration process. The subsequent step in its evolution will be to integrate it with tools for collaboration and knowledge discovery.

With the guidance and input of the MIT SSPARC team, several potential areas of enhancement have been identified for the application. The first involves elaborating the automated information extraction capabilities of SSPARCy. The application is only capable of extracting information it can unambiguously identify. This clear identification only can occur when consistent coding rules or practices are followed, whether enforced by the MATLAB syntax or by team consensus. Additional structure will be placed upon the code developed by the SSPARC design team. More specifically, the creation of a coding template will organize information in the MATLAB source files in such a fashion that rules can be established for the location of key information. For instance, information such as author, rationale, units, and valid range are all entered manually into the SSPARCy tool. This is necessary due to the lack of consistent standards from which rules can be derived for identifying the location of such information in source code files. If a template and accompanying standards were created, then this information could be clearly identifiable in the code and automatically extracted by the application.
Conversely, if the appropriate place for information in code files is established, then information from the SSPARCy application can be automatically inserted into source code files. For instance, a user could enter the rationale for a variable in the source code file, the application could extract it, present it to the user, allow the user to modify it using the application, and then update the source code file with the revised rationale. Such a capability would greatly enhance the automatic information extraction capabilities of SSPARCy. Furthermore, it would create a link and maintain consistency between the applications representation of information and the actual source code.

Another closely related improvement would update source code based upon errors determined by the check project feature of SSPARCy. If an error is determined by the application, the portion of the source code responsible for producing the error should be flagged. This capability would enable an integrator to quickly identify the precise location in source code that is creating integration errors.

A significant next generation feature desired for the application would be the association of information together using hyper links. When a reference to a different component of a simulation occurs when examining information about another component, a link could be created that when clicked would take the user to the information about the other component. Hyper linking functionality would make information navigation in the GUI more convenient for the user.

Another graphical enhancement would involve creation of a visual representation of the interaction between functions (in addition to the N-Squared View of section 4.2.3). This visual display would be similar to those present in the existing SSPARC tools and
shown in Figure 6. This feature would have an advantage over the existing SSPARC tools in that the visual representation would be automatically generated and updated.

The last identified potential augmentation would expand the functionality of the system to cater to multiple projects as well. Leveraging the capability of the application to record previous states, two or more applications could be compared against each other in meaningful ways. This feature would improve the scope of abilities of SSPARCy and increase its usefulness as a tool for analysis.

The implementation of these features would greatly improve the overall value of the SSPARCy application. It has the potential to become the primary tool used for the integration and analysis of MATLAB based projects, not just by SSPARC, but also by other design teams involved in similar endeavors.
8 Conclusion

The development of the SSPARCy tool began as a mandate to leverage the power of information technology to enhance the lean aerospace design process. After much input and discussion from all stakeholders, it evolved to take the form of a software integration support tool. It performs information extraction, logical representation of this information, historical recording, integration integrity verification and design rationale capture. Furthermore, a variety of graphical and visual features have been incorporated into the system to facilitate meaningful presentation of the variety of information stored by SSPARCy. The application offers a tremendous advancement over existing domain specific tools or grouping of disparate tools from other domains. Its diverse functionality is not matched by any peer application. SSPARCy, therefore, is a tool that can greatly improve the process of software integration.
References


[27] J. Vance. 'Design Issues in Virtual Reality." Associate Professor, Department of Mechanical Engineering, Iowa State University, August 1997.


10 Appendices

10.1 Back-End Architecture Code

All back-end architecture Code was written by Quincy Scott.

10.1.1 ClientIO

/* ClientIO.java
   *
   * Quincy Scott (quincy@mit.edu)
   * Spring 2001
   *
   * Description:
   * This class provides an interface for the client GUI to interact with the
   * underlying application. This interface will abstract away much of the
   * detail of the underlying implementation, including communication over a
   * network, if necessary.
   */

package SSPARC;

import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;
import java.io.IOException;

public class ClientIO {
    private Project p;
    public ClientIO() {
    }

    /*****************File methods**********************************/

   //Name of project and directory of code files
    public void createProject(String strName, String strDir) throws Exception {
        p = new Project(strName, strDir);
    }

    public void openProject(String strName) throws Exception {
        try {
            p = (Project) FileSaver.loadFileObject(strName);
        } catch (ClassNotFoundException cnfe) {
            throw new Exception("ClassNotFoundException: " +
                cnfe.getMessage());
        }
    }
}

120
public void saveProject(String strName) throws Exception {
    try {
        FileSaver.saveObjectInFile(p, strName);
    } catch (ClassNotFoundException cnfe) {
        throw new Exception("ClassNotFoundException: " + cnfe.getMessage());
    } catch (IOException ioe) {
        System.err.println(ioe.getMessage());
        throw new Exception("IOException: " + ioe.getMessage());
    }
}

/**********************************************Project methods**********************************************/
public String getProjectName() {
    return p.getName();
}

/***********************Constants*************************/
public String getConsName() {
    return p.getConsName();
}

public String getDesignName() {
    return p.getDesignName();
}

//returns an enumeration of the names of the constants of this project.
//each object of the enumeration is a String
public Enumeration getConstantsNames() {
    return p.getConstantsNames();
}

//returns a collection where each object is a Constant object.
public Enumeration getConstants() {
    return p.getConstants();
}

public Constant getConstant(String name) {
    return p.getConstant(name);  
}

public Vector getConstantsHist() {
    return p.getConstantsHist();
}

public void setConstantRat(String name, String rat) {
    p.setConstantRat(name, rat);
}
public void updateConstants() throws IOException {
    p.updateConstants();
}

//-----------------Design Variables-----------------

//view list of dvs
public Enumeration getDesignVarNames() {
    return p.getDesignVarNames();
}

//view list of dvs and values
public Enumeration getDesignVars() {
    return p.getDesignVars();
}

public DesignVar getDesignVar(String s) {
    return p.getDesignVar(s);
}

public Vector getDesignVarsHist() {
    return p.getDesignVarsHist();
}

//called when the user has updated the design file
public void updateDesignVars() throws IOException {
    p.updateDesignVars();
}

//returns an ErrorList of Error messages
public ErrorList checkProject() throws Exception {
    return p.check();
}

//--------------Functions----------------------------

//view a list of module names
public Enumeration getFunctionsNames() {
    return p.getFunctionsNames();
}

public void updateAll() throws Exception {
    p.updateAll();
}

public Enumeration getFunctions() {
    return p.getFunctions();
}

public Function getFunction(String s) {
    return p.getFunction(s);
}
public Vector getFunctionsHist() {
    return p.getFunctionsHist();
}

public void addFunction(String f) throws Exception {
    p.addFunction(f);
}

public void removeFunction(String f) throws Exception {
    p.removeFunction(f);
}

/****************Function methods******************/
//no longer exist in project. Must ask project for a specific //module and then interact with that module
/*
//view a module's rationale
public String getFunctionRat(String strFunctionName) {
    return p.getFunctionRat(strFunctionName);
}

//returns a table where the keys are the names of the inputs //and the values are InputOutput objects representing an input
public Hashtable getFunctionInputs(String strFunctionName) {
    return p.getFunctionInputs(strFunctionName);
}

//returns a table where the keys are the names of the outputs //and the values are InputOutput objects representing an output
public Hashtable getFunctionOutputs(String strFunctionName) {
    return p.getFunctionOutputs(strFunctionName);
}

//called when the GUI needs the rationale and date for an //Input/Output of a module
//returns an IO object representing the requested Input/Output
public InputOutput getFunctionIO(String strFunctionName, String strIO) throws Exception {
    return p.getFunctionIO(strFunctionName, strIO);
}

//returns a vector consisting of hashtables, each of which is //a table of a configuration of inputs for the module
public Vector getFunctionInputHist(String strFunctionName) {
    return p.getFunctionInputHist(strFunctionName);
}

//returns a vector consisting of hashtables, each of which is //a table of a configuration of inputs for the module
public Vector getFunctionOutputHist(String strFunctionName) {
    return p.getFunctionOutputHist(strFunctionName);
}
//returns a table of the modules called within the module passed //in as input.
public Hashtable getFunctionsCalled(String strFunctionName) {
  return p.getFunctionsCalled(strFunctionName);
}

//returns a vector consisting of hashtables, each of which is //a table of a configuration of inputs for the module
public Vector getFunctionsCalledHist(String strFunctionName) {
  return p.getFunctionsCalledHist(strFunctionName);
}

//called when the GUI needs a list of the fields of a structure
used
// by a module
public Vector getInputFields(String strFunctionName, String strInput) {
  return p.getInputFields(strFunctionName, strInput);
}

public Vector getOutputFields(String strFunctionName, String strOutput) {
  return p.getOutputFields(strFunctionName, strOutput);
}

//change a module's rationale
public void setFunctionRat(String strFunctionName, String strRat) {
  p.setFunctionRat(strFunctionName, strRat);
}

//change the rationale for an input to a module
public void setFunctionInputRat(String strFunctionName, String strInput, String strRat) {
  p.setFunctionInputRat(strFunctionName, strInput, strRat);
}

//change the rationale for an output to a module
public void setFunctionOutputRat(String strFunctionName, String strOutput, String strRat) {
  p.setFunctionOutputRat(strFunctionName, strOutput, strRat);
}*

//call this when a user has updated a module
public void updateFunction(String strFunctionName) throws Exception {
  p.updateFunction(strFunctionName);
}

}//closes class
10.1.2 Project

/* Project.java
   *
   * Quincy Scott (quincy@mit.edu)
   * Spring 2001
   *
   * Description:
   * The Project object is a representation of a given software project.
   * It is a grouping of functiones and methods to
   * access them.
   * It also contains a listing of the Design Variables, Constants and
   * their
   * values.
   *
   * See accompanying documentation for listing of assumptions.
   */

package SSPARC;

import java.util.Vector;
import java.util.Enumeration;
import java.util.StringTokenizer;
import java.io.*;

public class Project extends Variable {

    private Vector vFunctions; //list of FunctionLists
    private Vector vDesignVars; //list of DesignVarLists
    private Vector vConstants; //list of ConstantLists
    private Vector vErrors; //list of ErrorLists
    private String strDir; //directory of files
    private String strConsFile;
    private String strDesignFile;
    private String strCons;
    private String strDesign;

    public Project(String s, String dir) throws Exception {
        super(s, "", "", "", "", "");
        strDir = dir;

        vFunctions = new Vector();
        vFunctions.addElement(createFunctions());
        vConstants = new Vector();
        vConstants.addElement(createConstants());
        System.out.println("Made Constants...");
        vDesignVars = new Vector();
        vDesignVars.addElement(createDesignVars());
        System.out.println("Made Design Variables...");
        vErrors = new Vector();
    } //closes constructor

}
/************************Private methods
***********************

//used by constructor to create Functions table 
private FunctionList createFunctions() throws Exception {
    //initialize table
    FunctionList ml = new FunctionList();

    //gets a listing of all files in the directory
    File file = new File(strDir);
    File[] files = file.listFiles();
    Vector functionNames = new Vector();

    //goes through the list of files and saves the ones ending in .m 
    //that aren't Constants.m or Design.m
    for(int i = 0; i < files.length; i++) {
        File file1 = files[i];
        String fileName = file1.getName();
        StringTokenizer st = new StringTokenizer(fileName, Constants.FILE_NAME_EXT_SEPARATOR);
        if(st.countTokens() == 2) {
            String function = (String) st.nextToken();
            String ext = (String) st.nextToken();
            String extTest = ext.toLowerCase();
            String functionLow = function.toLowerCase();
            String consLow = Constants.CONSTANTS_FILENAME.toLowerCase();
            String designLow = Constants.DESIGNFILENAME.toLowerCase();

            if(extTest.equals(Constants.FILEEXTENSION) &&
               function.indexOf("#") != 0) {
                functionNames.addElement(fileName);
            }

            if(extTest.equals(Constants.FILEEXTENSION) &&
               functionLow.equals(consLow)) {
                strConsFile = fileName;
                strCons = function;
            }

            if(extTest.equals(Constants.FILEEXTENSION) &&
               functionLow.equals(designLow)) {
                strDesignFile = fileName;
                strDesign = function;
            }
        }
    }

    //creates a new function corresponding to each file name in the 
    //list
    //this assume that the file name and functionule name are the 
    //same
Enumeration ee = functionNames.elements();
while (ee.hasMoreElements()) {
    String fName = (String) ee.nextElement();

    Function mFunction = new Function(fName, "", "",
    functionNames.elements(), strDir);

    System.out.println(mFunction.getName());
    ml.add(mFunction.getName(), mFunction);
}

//now we've made all functions.
//they have each discovered the name used to call them
//now that we have this list of names, call update on each
//function so it can look for functions called using the call
names
Enumeration eF = ml.getAll();
while (eF.hasMoreElements()) {
    Function f = (Function) eF.nextElement();
    f.updateCalls(ml);
}

return ml;
} //closes createFunctions

private ConstantList createConstants() throws IOException {
    ConstantList cl = new ConstantList();

    FileReader frFunction = new
    FileReader(strDir+Constants.DIR_SEPARATOR+strConsFile);
    BufferedReader brFunction = new BufferedReader(frFunction);

    //go through each line of the Constants.m file
    for (String strLine = brFunction.readLine(); !(strLine == null);
        strLine = brFunction.readLine()) {

        //use this string to get the value
        String strCode = parseCommentsOut(strLine);
        //use this string to see if a constant assignment
        String strTest = parseStringsOut(strLine);
        //get non-commented portion of the line
        strTest = parseCommentsOut(strTest);

        int iCon = strTest.indexOf(Constants.CONSTANTS_CODE);
        int iAssign = strTest.indexOf(Constants.ASSIGN_VAL);

        //if function declaration, then skip line
        if (funcDeclar(strCode)) {
            continue;
        } else if ((iCon != -1) && (iCon < iAssign)) {
            //line contains constants and "="
            //see if "=" or another comparison involving =
            int assign[] = new int[4];
// see if the "=" is part of one of the following
assign[0] = strTest.indexOf("==");
assign[1] = strTest.indexOf(">=");
assign[2] = strTest.indexOf("<=");
assign[3] = strTest.indexOf("~=");

// find the first occurrence of the above
int lowest = strTest.length();
for (int i = 0; i < assign.length; i++) {
  if (assign[i] < lowest && assign[i] != -1) {
    lowest = assign[i];
  }
}

// if the index of "=" is less than any of the above then the line must have an assignment in it.
if ((iAssign < lowest) && (iAssign != -1)) {
  strTest = strTest.substring(0, iAssign);
} else {
  // otherwise, no assignment
  continue;
}

// now check for CONSTANTS, not a subsequence
if (iCon != 0) {
  String strFront = strTest.substring(iCon, iCon + 1);
  if (!strFront.equals(" ")) {
    continue;
  }
}

String strEnd = strTest.substring(iCon + Constants.CONSTANTS_CODE.length(),
iCon + Constants.CONSTANTS_CODE.length() + 1);

if (!strEnd.equals(Structures.STRUCT_FIELD_SEP_STR)) {
  continue;
}

// get the constant's name
int iBegInd = strTest.indexOf(Constants.CONSTANTS_CODE) + 10;
String strName = strTest.substring(iBegInd).trim();

if (strName.indexOf("(") != -1) {
  strName = strName.substring(0, strName.indexOf("("));
} else if (strName.indexOf("[") != -1) {
  strName = strName.substring(0, strName.indexOf("["));
} else if (strName.indexOf("{") != -1) {
  strName = strName.substring(0, strName.indexOf("{"));
}

strCode = strCode.substring(strCode.indexOf(Constants.ASSIGN_VAL) + 1);
//now get the value
//first, check to see if the value begins with a [ or ...
//which indicates a multi-line declaration of the value.
String strValFull = strCode;
String strVal = "";

if((strValFull.indexOf(Constants.ARRAY_OPEN) == -1) &&
(strValFull.indexOf(Constants.ARRAY_NEXT) == -1)) {
    strVal = strValFull.substring(0,
strValFull.indexOf(Constants.END_OF_LINE)).trim();
} else if((strValFull.indexOf(Constants.ARRAY_CLOSE) != -1) &&
(strValFull.indexOf(Constants.ARRAY_OPEN) != -1)) {
    int iEndIndex =
strValFull.indexOf(Constants.ARRAY_CLOSE);
    int iBegIndex =
strValFull.indexOf(Constants.ARRAY_OPEN);
    strVal = strValFull.substring(iBegIndex, iEndIndex+1);
} else {
    //if it is a multi-line declaration, then remember all
    //values and keep looping through new lines until you
    //reach
    //a "]") which indicates the closing of the value.
    while(true) {

        //first get the non-commented portion
        if(strValFull.indexOf(Constants.COMMENT) != -1) {
            int iEInd =
strValFull.indexOf(Constants.COMMENT);
            strValFull = strValFull.substring(0, iEInd);
        }

        //check to see if "...
        if(strValFull.indexOf(Constants.ARRAY_NEXT) != -1) {
            //if so, skip and read next line
            strValFull = brFunction.readLine();
        } else if(strValFull.indexOf(Constants.ARRAY_OPEN) !=
-1) {
            //see if "[", if so, remember line and read next
            strVal = strValFull;
            strValFull = brFunction.readLine();
        } else if(strValFull.indexOf(Constants.ARRAY_CLOSE) !=
-1) {
            //see if "]"
            //if so, read everything including the "]" and
            //appended it as a newline to the existing answer
            int iEndIndex =
strValFull.indexOf(Constants.ARRAY_CLOSE);
            strVal += \n"+strValFull.substring(0,
strValFull.indexOf(Constants.ARRAY_CLOSE));
    //stop reading this value
    break;
} else {
    //otherwise, add the current line as a new line
    //and read the next line
    strVal += \n"+strValFull;
    strValFull = brFunction.readLine();
}
//closes while
} //closes else for multi-line declarations

//if the name isn't "constant" than put that bad boy
//in the table of constants
if(!strName.equals(Constants.CONSTANTS_CODE)) {
    cl.add(strName, new Constant(strName, strVal, ",", ",");
} //closes if for a CONSTANTS declaration line
} //closes for loop looping through file

return cl;
} //closes createConstants

//creates new DVs and creates new DVs to replace ones whose values have
//changed.
private DesignVarList createDesignVars() throws IOException {
    DesignVarList cl = new DesignVarList();

    //just like the constants
    FileReader frFunction = new
    FileReader(strDir+Constants.DIR_SEPARATOR+strDesignFile);
    BufferedReader brFunction = new BufferedReader(frFunction);
    for(String strLine = brFunction.readLine(); !(strLine == null);
        strLine = brFunction.readLine()) {
        //use this string to get the value
        //use this string to get the value
        String strCode = parseCommentsOut(strLine);
        String strTest = parseStringsOut(strLine);
        //get non-commented portion of the line
        strTest = parseCommentsOut(strTest);

        int iCon = strTest.indexOf(Constants.DESIGN_CODE);
        int iAssign = strTest.indexOf(Constants.ASSIGN_VAL);

        //if function declaration, then skip line
        if(funcDeclar(strCode)) {
            continue;
        } else if((iCon != -1) && (iCon < iAssign)) {
            //line contains DESIGN and "="
            //see if "=" or another comparison involving =
            int assign[] = new int[4];

            //see if the "=" is part of one of the following
            assign[0] = strTest.indexOf("=");
            assign[1] = strTest.indexOf("=>");
            assign[2] = strTest.indexOf("==");
            assign[3] = strTest.indexOf(">=");

            //continue
            continue;
        } else {
            //see if the "=" is part of one of the following
            int assign[] = new int[4];

            //continue
            continue;
        }
    }

    //closes for loop
    return cl;
} //closes createDesignVars
assign[2] = strTest.indexOf("<=");
assign[3] = strTest.indexOf("~=");

//find the first occurrence of the above
int lowest = strTest.length();
for(int i=0;i<assign.length;i++) {
    if(assign[i] < lowest && assign[i] != -1) {
        lowest = assign[i];
    }
}

//if the index of "+=" is less than any of the above than
//the line must have an assignment in it.
if((iAssign < lowest) && (iAssign != -1)) {
    strTest = strTest.substring(0, iAssign);
} else {
    //otherwise, no assignment
    continue;
}

//now check for DESIGN, not a subsequence
if(iCon != 0) {
    String strFront = strTest.substring(iCon, iCon+1);
    if(!strFront.equals(" ")) {
        continue;
    }
}

String strEnd =
strTest.substring(iCon+Constants.DESIGN_CODE.length(),
iCon+Constants.DESIGN_CODE.length()+1);

if(!strEnd.equals( Constants.STRUCT_FIELD_SEP_STR)) {
    continue;
}

//get the constant's name
int iBegInd =
strTest.indexOf( Constants.DESIGN_CODE)+Constants.DESIGN_CODE.length()+1;
String strName = strTest.substring(iBegInd).trim();

if(strName.indexOf("(") != -1) {
    strName = strName.substring(0, strName.indexOf("("));
} else if(strName.indexOf("{") != -1) {
    strName = strName.substring(0, strName.indexOf("{"));
} else if(strName.indexOf("[") != -1) {
    strName = strName.substring(0, strName.indexOf("[");
}

strCode =
strCode.substring(strCode.indexOf( Constants.ASSIGN_VAL)+1);
//now get the value
//first, check to see if the value begins with a [ or ...
//which indicates a multi-line declaration of the value.
String strValFull = strCode;
String strVal = ""
if((strValFull.indexOf(\054Constants.ARRAY\054OPEN\054) == -1) &&
(strValFull.indexOf(\054Constants.ARRAY\054NEXT\054) == -1)) {
    strVal = strValFull.substring(0,
    strValFull.indexOf(\054Constants.END\054OF\054LINE\054)).trim();
} else if((strValFull.indexOf(\054Constants.ARRAY\054CLOSE\054) != -1) &&
(strValFull.indexOf(\054Constants.ARRAY\054OPEN\054) != -1)) {
    int iEndIndex =
    strValFull.indexOf(\054Constants.ARRAY\054CLOSE\054);
    int iBeginIndex =
    strValFull.indexOf(\054Constants.ARRAY\054OPEN\054);
    strVal = strValFull.substring(iBeginIndex, iEndIndex+1);
} else {
    //if it is a multi-line declaration, then remember all
    //values and keep looping through new lines until you
    //reach
    //a "]" which indicates the closing of the value.
    while(true) {

        //first get the non-commented portion
        if(strValFull.indexOf(\054Constants.COMMENT\054) != -1) {
            int iEndIndex =
            strValFull.indexOf(\054Constants.COMMENT\054);
            strValFull = strValFull.substring(0, iEndIndex);
        }

        //check to see if "..."
        if(strValFull.indexOf(\054Constants.ARRAY\054NEXT\054) != -1) {
            //if so, skip and read next line
            strValFull = brFunction.readLine();
        } else if(strValFull.indexOf(\054Constants.ARRAY\054OPEN\054) !=
-1) {
            //see if "]", if so, remember line and read next
            strVal = strValFull;
            strValFull = brFunction.readLine();
        } else if(strValFull.indexOf(\054Constants.ARRAY\054CLOSE\054)
!= -1) {
            //see if "]"
            //if so, read everything including the "]" and
            //appended it as a newline to the existing answer
            int iEndIndex =
            strValFull.indexOf(\054Constants.ARRAY\054CLOSE\054);
            strVal += \"\n\"+strValFull.substring(0,
            iEndIndex+1);
            //stop reading this value
            break;
        } else {
            //otherwise, add the current line as a new line
            //and read the next line
            strVal += \"\n\"+strValFull;
            strValFull = brFunction.readLine();
        }
    }
}
if(!strName.equals(Constants.DESIGN_CODE)) {
    cl.add(strName, new DesignVar(strName, strVal, "", "", ", ", ", ");
} //closes if for a DESIGN declaration line
} //closes for loop looping through file

return cl; //closes createDesignVars

public Enumeration getConstantsNames() {
    return ((ConstantList) vConstants.lastElement().getAllNames();
}

public Enumeration getConstants() {
    return ((ConstantList) vConstants.lastElement()).getAll();
}

public Vector getConstantsHist() {
    return vConstants;
}

public Constant getConstant(String s) {
    return ((ConstantList) vConstants.lastElement()).getConstant(s);
}

//If the constants have changed, then remember the old
//constants and update the new ones
public void updateConstants() throws IOException {
ConstantList clNewC = createConstants();
ConstantList clOldC = (ConstantList) vConstants.lastElement();

if(!clNewC.equals(clOldC)) {
    Enumeration eCons = clOldC.getAll();
    while(eCons.hasMoreElements()) {
        Constant con = (Constant) eCons.nextElement();
        if(clNewC.contains(con.getName())) {
            Constant cNew = clNewC.getConstant(con.getName());
            clNewC.remove(con.getName());
            Constant c = (Constant) con.clone();
            clNewC.add(c);
        }
    }
} //closes while
} //closes else for multi-line declarations

//if the name isn't "design" than put that bad boy
//in the table of dvs
if(!strName.equals(Constants.DESIGN_CODE)) {
    cl.add(strName, new DesignVar(strName, strVal, ", ", ", ", ", ", ");
} //closes if for a DESIGN declaration line
} //closes for loop looping through file

return cl; //closes createDesignVars

/****************************Public Methods and
helpers******************************/

//-----------------------------Constants Methods-----------------------------

public Enumeration getConstantsNames() {
    return ((ConstantList) vConstants.lastElement()).getAllNames();
}

public Enumeration getConstants() {
    return ((ConstantList) vConstants.lastElement()).getAll();
}

public Vector getConstantsHist() {
    return vConstants;
}

public Constant getConstant(String s) {
    return ((ConstantList) vConstants.lastElement()).getConstant(s);
}

public void updateConstants() throws IOException {
    ConstantList clNewC = createConstants();
    ConstantList clOldC = (ConstantList) vConstants.lastElement();

    if(!clNewC.equals(clOldC)) {
        Enumeration eCons = clOldC.getAll();
        while(eCons.hasMoreElements()) {
            Constant con = (Constant) eCons.nextElement();
            if(clNewC.contains(con.getName())) {
                Constant cNew = clNewC.getConstant(con.getName());
                clNewC.remove(con.getName());
                Constant c = (Constant) con.clone();
                clNewC.add(c);
            }
        }
    }
} //closes while
} //closes else for multi-line declarations

//if the name isn't "design" than put that bad boy
//in the table of dvs
if(!strName.equals(Constants.DESIGN_CODE)) {
    cl.add(strName, new DesignVar(strName, strVal, "", "", ", ", ", ");
} //closes if for a DESIGN declaration line
} //closes for loop looping through file

return cl; //closes createDesignVars

/****************************Public Methods and
helpers******************************/
c.setValue(cNew.getValue());
c1NewC.add(c.getName(), c);
}
}
vConstants.addElement(clNewC);
}

//------------------------------DV Methods----------------------------
---

public Enumeration getDesignVarNames() {
    return ((DesignVarList) vDesignVars.lastElement()).getAllNames();
}

public Enumeration getDesignVars() {
    DesignVarList dvl = (DesignVarList) vDesignVars.lastElement();
    return dvl.getAll();
}

public DesignVar getDesignVar(String s) {
    return ((DesignVarList) vDesignVars.lastElement()).getDV(s);
}

public Vector getDesignVarsHist() {
    return vDesignVars;
}

public void updateDesignVars() throws IOException {
    DesignVarList dvlNewC = createDesignVars();
    DesignVarList dvlOldC = (DesignVarList) vDesignVars.lastElement();

    if(!dvlNewC.equals(dvlOldC)) {
        Enumeration eDvs = dvlOldC.getAll();

        while(eDvs.hasMoreElements()) {
            DesignVar dv = (DesignVar) eDvs.nextElement();
            if(dvlNewC.contains(dv.getName())) {
                DesignVar dvNew = dvlNewC.getDV(dv.getName());
                dvlNewC.remove(dv.getName());
                DesignVar d = (DesignVar) dv.clone();
                d.setValue(dvNew.getValue());
                dvlNewC.add(d.getName(), d);
            }
        }

        vDesignVars.addElement(dvlNewC);
    }
}//closes updateDesignVars

//----------------------------Check Project-------------------------
134
/checks all function rules to ensure the following:
// 1. Check that all functions are called by another except for the start
// function (BTOS, ATOS, etc.) as specified by Constants.START_FUNC.
// 2. Check that all constants used appear in Constants structure.
// 3. Check that all dvs used appear in Design structure.
// 4. Check that all fields in Constants.m are used.
// 5. Check that all dvs in Design.m are used.
// 6. Check that input structures/fields are the output of another function
// 7. Check that all fields used for an input structure are defined as
// output fields for that same structure by another function.

//Returns an ErrorList of Error messages.
//also adds the ErrorList to the history of ErrorLists
public ErrorList check() throws Exception {

    //create list of error messages
    ErrorList el = new ErrorList();
    FunctionList ml = (FunctionList) vFunctions.lastElement();
    Enumeration eFunctions = ml.getAll();
    while (eFunctions.hasMoreElements()) {
        Function mFunction = (Function) eFunctions.nextElement();

        //NUMBER 1
        //check that all functions are called by another function
        //except for the start function as specified in the
        //Constants file (START_FUNC)
        if (!mFunction.getName().equals(Constants.START_FUNC)) {
            String name = mFunction.getName();
            Enumeration eFunctions2 = ml.getAll();
            while (eFunctions2.hasMoreElements()) {
                Function m2 = (Function) eFunctions2.nextElement();
                CallList c12 = m2.getCurrentCalled();
                if (c12.contains(name)) {
                    break;
                }
                if (!eFunctions2.hasMoreElements()) {
                    String msg = "function "+name+" not called by another
                    function";
                    el.addError(name, new Error(name, msg));
                }
            }
        }
    }

    //NUMBER 2

}
// Check that all constants used appear in Constants

// first, make sure Constants is an input.
InputList il = (InputList) mFunction.getCurrentInputs();
if(il.contains(Constants.CONSTANTS_CODE)) {
    Input inp = mFunction.getInput( Constants.CONSTANTS_CODE);
    InputFieldsList ifl = inp.getCurrentFields();
    Enumeration eFields = ifl.getAllNames();
    ConstantList conL = (ConstantList) vConstants.lastElement();
    while (eFields.hasMoreElements()) {
        String isf = (String) eFields.nextElement();
        if(!conL.contains(isf)) {
            String msg = "constant " + isf + " does not exist in constants file";
            el.addError(mFunction.getName(), new Error(mFunction.getName(), msg));
        }
    } // closes while
} // closes if

// NUMBER 3
// check that all dvs used appear in Design
il = (InputList) mFunction.getCurrentInputs();
if(il.contains( Constants.DESIGN_CODE)) {
    Input inp = mFunction.getInput( Constants.DESIGN_CODE);
    InputFieldsList ifl = inp.getCurrentFields();
    Enumeration eFields = ifl.getAllNames();
    DesignVarList dvl = (DesignVarList) vDesignVars.lastElement();
    while (eFields.hasMoreElements()) {
        String isf = (String) eFields.nextElement();
        if(!dvl.contains(isf)) {
            String msg = "design variable " + isf + " does not exist in design variable file";
            el.addError(mFunction.getName(), new Error(mFunction.getName(), msg));
        }
    } // closes while
} // closes if

// NUMBER 6
// check that all inputs are the outputs of another function
// assumes that output and input names are the same for every function if they correspond to the same entity.
InputList i16 = mFunction.getCurrentInputs();
Enumeration eInputs6 = i16.getAllNames();
while (eInputs6.hasMoreElements()) {
    String strInputName = (String) eInputs6.nextElement();
    FunctionList f16 = (FunctionList) vFunctions.lastElement();
    Enumeration efl16 = f16.getAll();
}
while(efl6.hasMoreElements()) {
    Function f6 = (Function) efl6.nextElement();
    if(f6.getName().equals(mFunction.getName())) {
        continue;
    }

    OutputList o16 = f6.getCurrentOutputs();
    if(o16.contains(strInputName)) {
        break;
    }

    if(!efl6.hasMoreElements()) {
        String msg = "input "+strInputName+" is not defined as output by any function";
        ei.addError(mFunction.getName(), new Error(mFunction.getName(), msg));
    }
}

//NUMBER 7
//check that all input fields used are defined as output fields by
//a function.
//assumes that the same name is used in all functions for a
//structure

InputList il7 = mFunction.getCurrentInputs();
Enumeration eI17 = il7.getAll();

//loop through the inputs
while(eI17.hasMoreElements()) {
    Input i7 = (Input) eI17.nextElement();
    String strIn = i7.getName();
    if(strIn.equals(Constants.CONSTANTS_CODE) ||
        strIn.equals(Constants.DESIGN_CODE)) {
        continue;
    }

    //get the fields for this input
    InputFieldsList ifl = i7.getCurrentFields();
    if(ifl.size() > 0) {
        Enumeration eFields = ifl.getAllNames();
        while(eFields.hasMoreElements()) {
            String isField = (String) eFields.nextElement();
            //loop through functions again

            FunctionList f172 = (FunctionList) vFunctions.lastElement();
            Enumeration eF72 = f172.getAll();
            while(eF72.hasMoreElements()) {
                Function f72 = (Function) eF72.nextElement();
                //now loop through the outputs for this function
                OutputList o17 = f72.getCurrentOutputs();
            }
        }
    }
}
if(ol7.contains(i7.getName())) {
    Output o7 = ol7.getOutput(i7.getName());
    OutputFieldsList ofl7 = o7.getCurrentFields();

    if(ofl7.contains(isField)) {
        break;
    }
}

// if you've gone through all the functions and none have
// the input structure as an output
if(!ef72.hasMoreElements()) {
    String msg = "input field "+isField+" of input structure "+i7.getName()+" is not defined in any output structure";
    el.addError(mFunction.getName(), new Error(mFunction.getName(), msg));
}
} // while loop for functions again...
} // closes while loop for input fields
} // closes if for input fields
} // closes while loop for inputs
} // closes loop for functions

// NUMBER 4
// Check that all constants in the constants file are used in another
// function.

// assumes constants are only used in a file if CONSTANTS are an
// input to the function
// also assumes that the presence of the characters CONSTANTS in
// the constants script is not a subsequence of characters of
// another struct.
ConstantList cl = (ConstantList) vConstants.lastElement();
Enumeration eConNames = cl.getAllNames();
while(eConNames.hasMoreElements()) {
    String strConName = (String) eConNames.nextElement();

    //get functions
    Enumeration eFunctions2 = ml.getAll();
    function_loop:
    while(eFunctions2.hasMoreElements()) {
        Function m2 = (Function) eFunctions2.nextElement();
        InputList il2 = m2.getCurrentInputs();
        if(il2.contains(Constants.CONSTANTS_CODE)) {
            Input i2 = il2.getInput(Constants.CONSTANTS_CODE);
            InputFieldsList ifl2 = i2.getCurrentFields();
            if(ifl2.contains(strConName)) {
                break;
            }
        } else if(m2.getName().toLowerCase().equals("constants")) {
            // if the function is constants, it is possible that the
// the constant being tested is only used internally with in
    // the constants script
    Output o = m2.getOutput(Parameters.CONSTANTS_CODE);
    OutputFieldsList of1 = o.getCurrentFields();
    Enumeration eOut = of1.getAll();
    int flag = 0;
    while (eOut.hasMoreElements()) {
      OutputStructField osf = (OutputStructField)
      eOut.nextElement();
      String val = osf.getValue();
      if (val.indexOf(Parameters.CONSTANTS_CODE + Constants.STRUCT_FIELD_SEP + strConName) == -1) {
        flag = 1;
        break;
      }
    }
    if (flag == 1) {
      break;
    }
  } // closes else if
  if (!eFunctions2.hasMoreElements()) {
    Enumeration eCons = cl.getAll();
    while (eCons.hasMoreElements()) {
      Constant cc = (Constant) eCons.nextElement();
      String strVal = cc.getValue();
      if (strVal.indexOf(Parameters.CONSTANTS_CODE + Constants.STRUCT_FIELD_SEP + strConName) != -1) {
        // then internally used constant
        break;
      } else if (!eCons.hasMoreElements()) {
        String msg = "constant " + strConName + " is not used in any function";
        // get the function name for constants... probably constants
        // or Constants
        Enumeration eDumb = ml.getAllNames();
        while (eDumb.hasMoreElements()) {
          String sl = (String) eDumb.nextElement();
          if (sl.toLowerCase().equals("constants")) {
            el.addError(sl, new Error(sl, msg));
            break;
          }
        }
      }
    }
  } //while going through functions2
//closes while going through constants names

//NUMBER 5
//Check that all design vars in the the design var file are used
//in some other function
//assumes design vars are only used in a file if DESIGN are an
//input to the function
DesignVarList dvl = (DesignVarList) vDesignVars.lastElement();
Enumeration eDvNames = dvl.getAllNames();
while(eDvNames.hasMoreElements()) {
    String strDvName = (String) eDvNames.nextElement();

    //get functionules
    Enumeration eFunctions2 = ml.getAll();
    while(eFunctions2.hasMoreElements()) {
        Function m2 = (Function) eFunctions2.nextElement();
        InputList il2 = m2.getCurrentInputs();
        if(il2.contains(Constants.DESIGN_CODE)) {
            Input i2 = il2.getInput(Constants.DESIGN_CODE);
            InputFieldsList if12 = i2.getCurrentFields();
            if(if12.contains(strDvName)) {
                break;
            }
        } else if(m2.getName().toLowerCase().equals("design")) {
            Output o = m2.getOutput(Constants.DESIGN_CODE);
            OutputFieldsList ofl = o.getCurrentFields();
            Enumeration eOut = ofl.getAll();
            int flag = 0;
            while(eOut.hasMoreElements()) {
                OutputStrucField osf = (OutputStrucField)
                eOut.nextElement();
                String val = osf.getValue();
                if(val.indexOf(Constants.DESIGNCODE + "." + strDvName) != -1) {
                    flag = 1;
                    break;
                }
            }
            if(flag == 1) {
                break;
            }
        }
    }
}

if(!eFunctions2.hasMoreElements()) {
    String msg = "design variable "+strDvName+" is not used in any function";

    //get the design var function name...probably Design or
    //design
    Enumeration eDumb = ml.getAllNames();
    while(eDumb.hasMoreElements()) {
        String s1 = (String) eDumb.nextElement();
        if(s1.toLowerCase().equals("design")) {
            e1.addError(s1, new Error(s1, msg));
        }
    }
}
break;
}
}
}
}   //while going through functions2
}   //closes while going through dv names

vErrors.addElement(el);
return el;

}   //closes check

public Vector getCheckHist() {
    return vErrors;
}

//-------------------------------Function Methods-------------------

public Enumeration getFunctionsNames() {
    FunctionList ml = (FunctionList) vFunctions.lastElement();
    return ml.getAllNames();
}

class Enumeration (FunctionList) {
    return ((FunctionList) vFunctions.lastElement()).getAll();
}

class Function getFunction(String s) {
    FunctionList ml = (FunctionList) vFunctions.lastElement();
    return ml.getFunction(s);
}

class Vector getFunctionsHist() {
    return vFunctions;
}

public void addFunction(String strFileName) throws Exception {
    FunctionList ml = (FunctionList) vFunctions.lastElement();
    Function function = new Function(strFileName, "", "",
    ml.getAllNames(), strDir);
    ml.add(function.getName(), function);
    updateAll();
}

class void removeFunction(String functionName) throws Exception {
    FunctionList ml = (FunctionList) vFunctions.lastElement();
    ml.remove(functionName);
    updateAll();
}

class void updateAll() throws Exception {
    FunctionList ml = (FunctionList) vFunctions.lastElement();
}
Enumeration eFunctions = ml.getAll();

while(eFunctions.hasMoreElements()) {
    ((Function)
      eFunctions.nextElement().update(ml.getAllNames());
}

//-----------------------------Individual Function Methods-----------------------------

public void updateFunction(String strFunctionName) throws Exception
{
    FunctionList ml = (FunctionList) vFunctions.lastElement();
    Function m = ml.getFunction(strFunctionName);
    m.update(((FunctionList)
      vFunctions.lastElement()).getAllNames());
}

//-------------------------------Parse Code---------------------------------------------

public String parseStringsOut(String strLine) {

    String strCode = "";
    int iString = strLine.indexOf(""");

    //get rid of strings
    //first see if the line even contains a ' before a %

    StringTokenizer st;
    if((iString != -1) \&\& ((iString < strLine.indexOf("%")) ||
     (strLine.indexOf("%") == -1))) {
        //break line up by '
        st = new StringTokenizer(strLine, ",", true);
        //if only one ' then it can't contain a string

        String oldest = "";
        String oldestNo = "";
        String old = "";
        String oldNo = "";
        String current = "";
        String currentNo = "";

        while(st.countTokens() > 0) {

            currentNo = ((String) st.nextToken());
            current = currentNo.trim();

            if(oldest.equals("") \&\& current.equals("") {

                if(old.startsWith("+") || old.startsWith("-") ||
                old.startsWith("*") || old.startsWith("^") || old.startsWith("\") ||
                old.startsWith("/\") || old.startsWith("==") || old.startsWith("<") ||

...
old.startsWith(">") || old.startsWith("&") || old.startsWith("|") ||
old.startsWith("~") {

    if(old.endsWith("+") || old.endsWith("-") ||
old.endsWith("*") || old.endsWith("^") || old.endsWith("\") ||
old.endsWith("/") || old.endsWith("==") || old.endsWith("<") ||
old.endsWith(">") || old.endsWith("&") || old.endsWith("|") ||
old.endsWith("~") || old.endsWith(""") || old.endsWith("[") ||
old.endsWith("{") {
        strCode = strCode+oldestNo+oldNo;
        oldest = "";
        oldestNo = "";
        old = current;
        oldNo = currentNo;
    } else {

        // include
        strCode = strCode + oldestNo+oldNo+currentNo;
        oldest = "";
        oldestNo = "";
        old = "";
        oldNo = "";
    }
}

    continue;
} else {

    // string
    oldest = "";
    oldestNo = "";
    old = "";
    oldNo = "";
    current = "";
    currentNo = "";
}

if(st.countTokens() == 0) {
    strCode = strCode+oldestNo+oldNo+currentNo;
    continue;
}

strCode = strCode + oldestNo;

    oldest = old;
    oldestNo = oldNo;
    old = current;
    oldNo = currentNo;

}// closes while

} else {// closes if ' before %

    return strLine;
}

return strCode;
private String parseCommentsOut(String strLine) {
    String strCode;
    if(strLine.indexOf(Constants.COMMENT) != -1) {
        int iEndInd = strLine.indexOf(Constants.COMMENT);
        strCode = strLine.substring(0, iEndInd);
    } else {
        strCode = strLine;
    }
    return strCode;
}

private boolean funcDeclar(String strLine) {
    strLine = strLine.trim();
    int iIndex = strLine.indexOf(Constants.FUNCDECLAR);
    if(iIndex == -1) {
        return false;
    } else {
        strLine = strLine.trim();
        // if so, and the name is not the beginning of the line
        if(iIndex != 0) {
            // if the function name is not the first char of the line
            // then get the preceding char
            String c = strLine.substring(iIndex-1, iIndex);
            // check to make sure an op,
            // done to ensure name is not a substring
            if(!c.equals(" ")) {
                return false;
            }
        }
        // just check the last character
        if((iIndex+Constants.FUNCDECLAR.length()-1) < (strLine.length()-1)) {
            String cEnd = strLine.substring(iIndex+Constants.FUNCDECLAR.length(),
                iIndex+Constants.FUNCDECLAR.length()+1);
            if((cEnd.equals(" ")) || (cEnd.equals("["))) {
                return true;
            }
        } else {
            // name is the last string of the line
            return true;
        }
    }
}
} //closes else
if(strLine.indexOf("time1") != -1) {
    System.out.println("incorrect ending");
}

return false;
}

public String getConsName() {
    return strCons;
}

public String getDesignName() {
    return strDesign;
}

} //closes class
10.1.3 Function

/* Function.java
 * Quincy Scott (quincy@mit.edu)
 * Spring 2001
 * Description:
 * Represents a single module of a project. Groups the inputs, outputs, and
 * functions/script (does not track subfunctions) called with in the code of
 * the module. Also stores the histories
 * and rationales for the inputs, outputs and modules called.
 * See accompanying documentation for a description of assumptions made */

package SSPARC;

import java.util.*;
import java.io.*;

public class Function extends Variable {

    private String strDir; //directory where modules reside
    private String strFileN;
    private Vector vInputs; //history of inputlists
    private Vector vOutputs; //history of outputlists
    private Vector vCalls; // history of calllists

    public Function(String name, String rat, String author, Enumeration eFunctionNames, String sDir) throws Exception {
        super(name.substring(0, name.indexOf(Constants.FILE_NAME_EXT_SEPARATOR)), "", "", "", rat, author);
        strFileN = name;

        strDir = sDir;
        vInputs = new Vector();
        vOutputs = new Vector();
        vCalls = new Vector();

        //sets up the list of the current inputs, outputs and mods called
        setup(eFunctionNames);
        System.out.println("setup");
        setupFields();
        System.out.println("setupFields");
        setupOutputFields();
        System.out.println("setupOutputs");
    }//closes constructor
/*******************************Private
Methods******************************

//First finds the function declaration line to determine outputs
and inputs
//Next it scans the code of the module to see if any calls are made
to
//modules with the name in the module name list.
private void setup(Enumeration eFunctionNames) throws Exception {
    //open module file
    FileReader frFunction = new
    FileReader(strDir+Constants.DIR_SEPARATOR+strFileN);
    BufferedReader brFunction = new BufferedReader(frFunction);

    //make list of mods/functions called
    CallList cl = new CallList();
    boolean bFuncDeclare = false;

    //loop through each line of the file
    for(String strLine = brFunction.readLine(); !(strLine ==
null);strLine = brFunction.readLine()) {
        String strCode = parseStringsOut(strLine);

        strCode = parseCommentsOut(strCode);

        //check to see if it is the function declaration line
        //only process the first function declaration line (ignores
        //subfunctions)
        if(!bFuncDeclare) &
            funcDeclar(strCode)) {

            bFuncDeclare = true;

            //see if this function declaration contains a continuation
            //symbol "...
            if(strCode.indexOf(Con	
Constants.ARRAY_NEXT) != -1) {

                //if continuation, then must be of outputs or inputs
                //if so, read in successive lines until you reach a
                //"}" or ")"
                //append to the first line, minus the "..."

                // strCode =
                strCode.substring(strCode.indexOf(Con	
stants.ARRAY_NEXT)+Constants.ARRAY
_NEXT.length()));

                strCode = strCode.substring(0,
                strCode.indexOf(Con	
stants.ARRAY_NEXT));

                while(true) {
                    strLine = brFunction.readLine();
                    String strTmp;
                    if(strLine.indexOf(Con	
stants.COMMENT) == -1) {
                        strTmp = strLine;
                    } else {

147
int iEndInd = strLine.indexOf(_constants.COMMENT);
strTmp = strLine.substring(0, iEndInd).trim();

strCode = strCode + strTmp;

if((strTmp.indexOf(_constants.ARRAY_CLOSE) != -1) ||
(strTmp.indexOf(_constants.FUNC_CLOSE) != -1)) {
    break;
}

// closes while

// let's make the Function name the name in the declaration line

// get the index of the "=" and the "(" int iBeg = strCode.indexOf(_constants.ASSIGN_VAL);
int iEnd = strCode.indexOf(_constants.FUNC_OPEN);

String newName;

// see if the line contains a "=",

// assumes a "(" can never proceed a "=" in the declaration
// function declaration line of
if((iBeg != -1) && (iEnd != -1)) {
    // get everything from "=" to "(
    newName = strCode.substring(iBeg+1, iEnd).trim();
} else if((iBeg != -1) && (iEnd == -1)) {
    // means "=" but no "(
    // get everything after "=
    newName = strCode.substring(iBeg+1).trim();
} else if((iEnd != -1) && (iBeg == -1)) {
    // means "(" but no "=
    // get everything after "function" until "(
    newName = strCode.substring(strCode.indexOf(_constants.FUNC_DECLAR)+Constants.FUNC_DECLAR.length(), iEnd).trim();
} else {
    // means no "(" and no "="
    // get it all after "function"
    newName = strCode.substring(strCode.indexOf(_constants.FUNC_DECLAR)+Constants.FUNC_DECLAR.length()).trim();
}

setName(newName);

vInputs.addElement(setupInputs(strCode));

vOutputs.addElement(setupOutputs(strCode));
} else {

// closes else
// closes for loop

// if no input tables in the list of input tables,
if(vInputs.size() < 1) {
    // then script
    vInputs.addElement(new InputList());
}
private InputList setupInputs(String strLine) {
    //create a list for all the inputs
    InputList il = new InputList();

    //now for the inputs...
    int iBeginOut = strLine.indexOf(Constants.FUNC_OPEN);
    int iEndOut = strLine.indexOf(Constants.FUNC_CLOSE);
    //get inputs
    //if no "(" on the function declaration line or
    //if "(" and ")" are consecutive characters
    if(iBeginOut == -1 || (iEndOut == (iBeginOut + 1))) {
        //no inputs, a script
        //return an empty hashtable
        return il;
    }

    //if there are inputs, let's get them.
    String strInputs = strLine.substring(iBeginOut+1, iEndOut);
    //separate the inputs by the appearance of ","
    StringTokenizer stInputs = new StringTokenizer(strInputs,
        Constants.FUNC_INPUTSEP);

    //loop through inputs
    while(stInputs.hasMoreTokens()) {
        String strInput = (String) stInputs.nextToken();
        strInput = strInput.trim();
        //add each input to the input list
        il.add(strInput, new Input(strInput, "", "", ",", "", ",", "", "");
    } //closes while
    return il;
} //closes setupInputs

private OutputList setupOutputs(String strLine) {
    //create a table for all the outputs
    OutputList ol = new OutputList();

    //make sure the line contains a "="
    if(strLine.indexOf(Constants.ASSIGN_VAL) != -1) {

int iBeginOut;
int iEndOut;

//see if the line contains a "[
if(strLine.indexOf(Constants.FUNC_OUTPUT_OPEN) != -1) {
    //if so, assume a "]" on the line as well
    //get the indices of the "[" and "]"
    iBeginOut = strLine.indexOf(Constants.FUNC_OUTPUT_OPEN);
    iEndOut = strLine.indexOf(Constants.FUNC_OUTPUT_CLOSE);
} else {
    //if no "[", then get everything from the end of
    //"function" to "=
    iBeginOut = strLine.indexOf(Constants.FUNC_DECLAR)+Constants.FUNC_DECLAR.length();
    iEndOut = strLine.indexOf(Constants.ASSIGN_VAL);
}

//gets the outputs
String strOutputs = strLine.substring(iBeginOut+1, iEndOut);
//divide up the outputs by ","
StringTokenizer stOutputs = new StringTokenizer(strOutputs,
    Constants.FUNC_OUTPUT_SEP);

//loop through outputs
while(stOutputs.hasMoreTokens()) {
    String strOutput = (String) stOutputs.nextToken();
    strOutput = strOutput.trim();
    //add each output to the output hashtable
    ol.add(strOutput, new Output(strOutput, ".", ",", ",", ","));
} //closes while
return ol;
//closes setupOutputs

private Call setupCall(Enumeration eFunctions, String strLine) {
    //go through all of the possible function/script names
    while(eFunctions.hasMoreElements()) {
        String strFunctionName = (String) eFunctions.nextElement();
        //strFunctionName = strFunctionName+Constants.FUNC_OPEN;

        strLine = strLine.trim();
        //see if this line contains "name"
        int iIndex = strLine.indexOf(strFunctionName);
        if(iIndex != -1) {
            //if so, and the name is not the beginning of the line
            if(iIndex != 0) {
//if the function name is not the first char of the
//then get the preceding char
char c = strLine.charAt(iIndex-1);

//check to make sure an op,
//done to ensure name is not a substring
if((c == '(') || (c == '[') || (c == ')') || (c == ']') || (c == '+') || (c == '-') || (c == '*') || (c == '/') || (c == '<') || (c == '>') || (c == '&') || (c == '=') || (c == '-') || (c == ',') || (c == ';') || (c == '}') || (c == '[') || (c == '<') || (c == '>') || (c == ','))
{
    continue;
}

//name starts the line.
//just check the last character

if(iIndex != (strLine.length()-1)) {
    char cEnd = strLine.charAt(iIndex+strFunctionName.length());
    if((cEnd == ';') || (cEnd == '}') || (cEnd == ':') || (cEnd == ')') || (cEnd == ']') || (cEnd == '+') || (cEnd == '-') || (cEnd == '*') || (cEnd == '/') || (cEnd == '<') || (cEnd == '>') || (cEnd == '&') || (cEnd == '=') || (cEnd == ',') || (cEnd == ';'))
    {
        return new Call(strFunctionName, "", "", "", "", "");
    }
} else {
    //name is the last string of the line

    return new Call(strFunctionName, "", "", "", "", "");
}

//closes if

//closes while

return null;

private void setupFields() throws Exception {

    //first, let's tell all the input objects that we
    //are updating their fields
    Enumeration eInputs = ((InputList)vInputs.lastElement()).getAll();
    while (eInputs.hasMoreElements()) {
        ((Input) eInputs.nextElement()).updateFields();
    }

    //open file
    FileReader frFunction = new FileReader(strDir+Constants.DIR_SEPARATOR+strFileN);
    BufferedReader brFunction = new BufferedReader(frFunction);
//loop through each line of the module file
for(String strLine = brFunction.readLine(); !(strLine == null); strLine = brFunction.readLine()) {
    String strCode = parseStringsOut(strLine);
    strCode = parseCommentsOut(strCode);

    //check to make sure not a function declaration line
    if(funcDeclar(strCode)) {
        continue;
    }

    //get the portion of the code following an "="
    if(strCode.indexOf(Statics.ASSIGNVAL) != -1) {
        int iAssign = strCode.indexOf(Statics.ASSIGNVAL);
        int assign[] = new int[4];

        //see if the "=" is part of one of the following
        assign[0] = strCode.indexOf("=");
        assign[1] = strCode.indexOf(">=");
        assign[2] = strCode.indexOf("<=");
        assign[3] = strCode.indexOf("-=");

        //find the first occurrence of the above
        int lowest = strCode.length();
        for(int i=0;i<assign.length;i++) {
            if(assign[i] < lowest && assign[i] != -1) {
                lowest = assign[i];
            }
        }

        //if the index of "=" is less than any of the above than
        //the line must have an assignment in it.
        if((iAssign < lowest) && (iAssign != -1)) {
            //get all code after the assignment
            strCode = strCode.substring(iAssign+1);
        }
        //otherwise, no assignment
    }

    //check to see if the line uses an input
    InputList il = (InputList) vInputs.lastElement();
    Enumeration e = il.getAllNames();

    //loop through the names of the inputs
    while(e.hasMoreElements()) {
        String strThis = strCode;
        strThis = strThis.trim();

        //get input name
        String strName = (String) e.nextElement();
        String strTest = strName + Statics.STRUCTFIELDSEP;

        //see if this line contains an input structure's name + "." 
        if(strThis.indexOf(strTest) != -1) {

        }
// while there are still characters to examine in this
// string...
for(int iInd=0;strThis.length() > 0; ) {
    // makes sure input structure's name is not a
    if(strThis.indexOf(strTest) == 0) {
        // substring
        strThis.substring(strThis.indexOf(strTest)-1,
        strThis.indexOf(strTest));
        if(!(c.equals("+") || c.equals("-") ||
        c.equals("*") || c.equals("/") || c.equals("\") || c.equals("=") ||
        c.equals("<") || c.equals(">") || c.equals("," || c.equals("-") ||
        c.equals("&") || c.equals("1") || c.equals("(" || c.equals("[") ||
        c.equals("{") || c.equals("") || c.equals(":"))) {
            // implies not a valid use
            // examine the rest of the line
            strThis = strThis.substring(strThis.indexOf(strTest)+strTest.length());
            continue;
        }
    } // closes else

    // valid occurrence of name
    // get the portion from the beginning of the name to
    // end of the line
    strThis = strThis.substring(strThis.indexOf(strTest));

    int iBegInd;
    int iEndInd;

    // thus a structure
    // then get the name of the field.
    // start with the point right after the .
    iBegInd = strThis.indexOf(strName)+strName.length()+1;
    iEndInd = 0;

    // guess all possible endings for the name of the
    field
    int endings[] = new int[30];
    endings[0] = strThis.indexOf(";", iBegInd);
    endings[1] = strThis.indexOf("+", iBegInd);
    endings[2] = strThis.indexOf("-", iBegInd);
    endings[3] = strThis.indexOf("*", iBegInd);
    endings[4] = strThis.indexOf("\", iBegInd);
    endings[5] = strThis.indexOf("\", iBegInd);
    endings[6] = strThis.indexOf("/", iBegInd);
    endings[7] = strThis.indexOf("=", iBegInd);
    endings[8] = strThis.indexOf("~", iBegInd);
endings[9] = strThis.indexOf("<", iBegInd);
endings[10] = strThis.indexOf(">", iBegInd);
endings[11] = strThis.indexOf("\", iBegInd);
endings[12] = strThis.indexOf("\", iBegInd);
endings[13] = strThis.indexOf("\", iBegInd);
endings[14] = strThis.indexOf("\", iBegInd);
endings[15] = strThis.indexOf("\", iBegInd);
endings[16] = strThis.indexOf("\", iBegInd);
endings[17] = strThis.indexOf("\", iBegInd);
endings[18] = strThis.indexOf("\", iBegInd);
endings[19] = strThis.indexOf("\", iBegInd);
endings[20] = strThis.indexOf("\", iBegInd);
endings[21] = strThis.indexOf("\", iBegInd);
endings[22] = strThis.indexOf("\", iBegInd);
endings[23] = strThis.indexOf("\", iBegInd);
endings[24] = strThis.indexOf("\", iBegInd);
endings[25] = strThis.indexOf("\", iBegInd);
endings[26] = strThis.indexOf("\", iBegInd);
endings[27] = strThis.indexOf("\", iBegInd);
endings[28] = strThis.indexOf("\", iBegInd);
endings[29] = strThis.indexOf("\", iBegInd);

//see which is the ending for the name by

choosing smallest

//index greater than the beginning index.
for(int i = 0; i<endings.length; i++) {
    if((endings[i]>iBegInd)) {
        if(iEndInd == 0) {
            iEndInd = endings[i];
        } else if(endings[i]<iEndInd) {
            iEndInd = endings[i];
        }
    }
}

//if none other the above terminations, then
//must be a line with no termination
if(iBegInd > iEndInd) {
    iEndInd = strThis.length();
}

//get the name
String strField = strThis.substring(iBegInd,
iEndInd).trim();

//now check to see if the field existed in the
//previous list of fields
//get the input
Input in = il.getInput(strName);

//get its history of fields
Vector vHist = in.getFieldsHistory();
//if it has a history
if(vHist.size() > 1) {
    //then get the field list before the current
one
    InputFieldsList iFields = (InputFieldsList)
vHist.elementAt(vHist.size()-2);
// see if the list contains the field we are adding
if (iFields.contains(strField)) {
    // if so, get it and add its clone to the current list
    InputStrucField isf = (InputStrucField) iFields.get(strField);
    in.addField((InputStrucField) isf.clone());
} else {
    // if not, then add it by making a new one
    in.addField(strField);
}
else {
    // if no previous lists, then just add it
    in.addField(strField);
}

// if another occurrence of the same name occurs on this line
if (strThis.indexOf(strName, iEndInd) != -1) {
    strThis = strThis.substring(strThis.indexOf(strName, iEndInd), strThis.length());
} else {
    strThis = "";
}

} // closes for that loops through the chars of the line.
} // closes if seeing if the line contains an instance of the name

} // closes while loop through inputs

} // closes for looping through file
} // closes setupFields

private void setupOutputFields() throws Exception {
    Enumeration eOutputs = ((OutputList) vOutputs.lastElement()).getAll();
    while (eOutputs.hasMoreElements()) {
        ((Output) eOutputs.nextElement()).updateFields();
    }

    // open file
    FileReader frFunction = new FileReader(strDir + Constants.DIR_SEPARATOR + strFileN);
    BufferedReader brFunction = new BufferedReader(frFunction);

    for (String strLine = brFunction.readLine(); !(strLine == null); strLine = brFunction.readLine()) {
        String strCode = parseStringsOut(strLine);
        strCode = parseCommentsOut(strCode);
    }
//check to make sure not a function declaration line
if(funcDeclar(strCode)) {
    continue;
}

//see if the line doesn't contain a "="
if(strCode.indexOf( Constants.ASSIGN_VAL) == -1) {
    continue;
} else {
    int iAssign = strCode.indexOf( Constants.ASSIGN_VAL);
    int assign[] = new int[4];
    assign[0] = strCode.indexOf("==");
    assign[1] = strCode.indexOf(">=");
    assign[2] = strCode.indexOf("<=");
    assign[3] = strCode.indexOf("-=");
    int lowest = strCode.length();
    for(int i=0;i<assign.length;i++) {
        if(assign[i] < lowest && assign[i] != -1) {
            lowest = assign[i];
        }
    }
    //if the index of "=" is less than any of the above than
    //the line must have an assignment in it.
    if((iAssign < lowest) && (iAssign != -1)) {
        //get all code before the assignment
        strCode = strCode.substring(0, iAssign);
    } else {
        //otherwise, no assignment
        continue;
    }
}

Enumeration e = ((OutputList) vOutputs.lastElement()).getAllNames();

while(e.hasMoreElements()) {
    String strName = (String) e.nextElement();
    if(strCode.indexOf(strName) == -1) {
        continue;
    }
    String strField = "";
    //check to see if this is a field of a structure
    if(strCode.indexOf( Constants.STRUCT_FIELD_SEP) != -1) {
        int iBeg = strCode.indexOf(strName)+strName.length()+1;
    }
}
if (strCode.indexOf(Constants.FUNC_OPEN, iBeg) != -1) {
    strField = strCode.substring(iBeg,
    strCode.indexOf("\"").trim());
} else if (strCode.indexOf("\{"") != -1) {
    strField = strCode.substring(iBeg,
    strCode.indexOf("\{"").trim());
} else if (strCode.indexOf("\[") != -1)
    strField = strCode.substring(iBeg).trim();
else {
    strField = strCode.substring(iBeg).trim();
}

OutputList ol = (OutputList) vOutputs.lastElement();
Output out = ol.getOutput(strName);
Vector vHist = out.getFieldsHistory();
if (vHist.size() > 1) {
    OutputFieldsList oFields = (OutputFieldsList)
    vHist.elementAt(vHist.size()-2);
    if (oFields.contains(strField)) {
        OutputStrucField osf =
        oFields.getField(strField);
        out.addField((OutputStrucField) osf.clone());
    } else {
        out.addField(strField);
    }
} else {
    out.addField(strField);
}
}//closes if
}
}//closes while
}//closes for
}//closes setupOutputFields

="/************************Public
Methods**************************/

public InputList getCurrentInputs() {
    return (InputList) vInputs.lastElement();
}

public OutputList getCurrentOutputs() {
    return (OutputList) vOutputs.lastElement();
}

public CallList getCurrentCalled() {
    return (CallList) vCalls.lastElement();
}

public Vector getInputsHist() {
    return vInputs;
}
public Vector getOutputsHist() {
    return vOutputs;
}

public Vector getCalledHist() {
    return vCalls;
}

public Input getInput(String s) {
    InputList il = (InputList) vInputs.lastElement();
    return (Input) il.get(s);
}

public Output getOutput(String s) {
    OutputList ol = (OutputList) vOutputs.lastElement();
    return (Output) ol.get(s);
}

public Call getCall(String s) {
    CallList cl = (CallList) vCalls.lastElement();
    return (Call) cl.get(s);
}

public void updateCalls(FunctionList f1) throws Exception {

    //open file
    FileReader frFunction = new FileReader(strDir+Constants.DIR_SEPARATOR+strFileN);
    BufferedReader brFunction = new BufferedReader(frFunction);
    CallList cl = new CallList();

    //loop through each line of the file
    for(String strLine = brFunction.readLine(); ! (strLine == null) ; strLine = brFunction.readLine()) {
        String strCode = parseStringsOut(strLine);
        //get non-commented portion of the line
        strCode = parseCommentsOut(strCode);
        //if(strLine.indexOf(Constants.COMMENT) != -1) {
        //    int iEndInd = strLine.indexOf( Constants.COMMENT);
        //    strCode = strLine.substring(0, iEndInd);
        //} else {
        //    strCode = strLine;
        //}

        if(strCode.length() == 0) {
            continue;
        }

        //check to see if it is the function declaration line
        if(funcDeclar(strCode)) {
            continue;
        } else {


// might make call to another module
Call c = setupCall(fl.getAllNames(), strCode);
if(c != null) {
    cl.add(c.getName(), c);
} // closes if
} // closes else
}// closes for loop

if(cl != null) {
    if(vCalls.size() > 0) {
        CallList clOld = (CallList) vCalls.lastElement();
        if(!clOld.equals(cl)) {
            Enumeration eCalls = clOld.getAll();
            while(eCalls.hasMoreElements()) {
                Call call = (Call) eCalls.nextElement();
                if(cl.contains(call.getName())) {
                    cl.remove(call.getName());
                    cl.add(call.getName(), (Call) call.clone());
                }
            }
            vCalls.addElement(cl);
        } else {
            vCalls.addElement(cl);
        }
    }
} // closes if

public void update(Enumeration eFunctionNames) throws Exception {
    Vector vFunctionNames = new Vector();
    while(eFunctionNames.hasMoreElements()) {
        vFunctionNames.addElement((String) eFunctionNames.nextElement());
    }
    // open file
    FileReader frFunction = new FileReader(strDir+Constants.DIR_SEPARATOR+strFileN);
    BufferedReader brFunction = new BufferedReader(frFunction);
    InputList il = null;
    OutputList ol = null;
    CallList cl = new CallList();

    // loop through each line of the file
    for(String strLine = brFunction.readLine(); !(strLine == null); strLine = brFunction.readLine()) {
        String strCode = parseStringsOut(strLine);
        
        // rest of the code
    } // closes for loop
} // closes method
//get non-commented portion of the line
if(strCode.indexOf(Constants.COMMENT) != -1) {
    int iEndInd = strLine.indexOf(Constants.COMMENT);
    strCode = strLine.substring(0, iEndInd);
} else {
    strCode = strLine;
}

//check to see if it is the function declaration line
if(funcDeclar(strCode)) {
    il = setupInputs(strCode);
    ol = setupOutputs(strCode);
} else {
    //might make call to another module
    Call c = setupCall(vFunctionNames.elements(), strCode);
    if(c != null) {
        cl.add(c.getName(), c);
    } //closes if
    } //closes else
} //closes for loop

//now check to see if the current states are different than the previous
if(il != null) {
    //get the table of the last set of inputs
    InputList ilOld = (InputList) vInputs.lastElement();

    //see if they are different
    if(!ilOld.equals(il)) {
        //if so, copy the same inputs from the old list to the new
        Enumeration eInputs = ilOld.getAll();
        while(eInputs.hasMoreElements()) {
            Input in = (Input) eInputs.nextElement();
            if(il.contains(in.getName())) {
                il.remove(in.getName());
                il.add(in.getName(), (Input) in.clone());
            }
        }

        vInputs.addElement(il);
    }
} //closes if

setupFields();

if(ol != null) {
    OutputList olOld = (OutputList) vOutputs.lastElement();
    if(!olOld.equals(ol)) {

Enumeration eOutputs = olOld.getAll();

while (eOutputs.hasMoreElements()) {
    Output out = (Output) eOutputs.nextElement();
    if (ol.contains(out.getName())) {
        ol.remove(out.getName());
        ol.add(out.getName(), (Output) out.clone());
    }
}
//closes if

vOutputs.addElement(ol);
}

setupOutputFields();

if (cl != null) {
    if (vCalls.size() > 0) {
        CallList clOld = (CallList) vCalls.lastElement();
        if (!clOld.equals(cl)) {
            Enumeration eCalls = clOld.getAll();

            while (eCalls.hasMoreElements()) {
                Call call = (Call) eCalls.nextElement();
                if (cl.contains(call.getName())) {
                    cl.remove(call.getName());
                    cl.add(call.getName(), (Call) call.clone());
                }
            }
            vCalls.addElement(cl);
        }
    } else {
        vCalls.addElement(cl);
    }
}
//closes if

} //closes update

public Object clone() {
    return this;
} //closes clone

public String parseStringsOut(String strLine) {
    String strCode = "";
    int iString = strLine.indexOf('"");

    //get rid of strings
    //first see if the line even contains a ' before a %
StringTokenizer st;
if((iString != -1) && ((iString < strLine.indexOf("%")) ||
(strLine.indexOf("%") == -1))) {

    //break line up by '
    st = new StringTokenizer(strLine, "', true);
    //if only one ' then it can't contain a string

    String oldest = ""
    String oldestNo = ""
    String old = ""
    String oldNo = ""
    String current = ""
    String currentNo = ""

    while(st.countTokens() > 0) {
        currentNo = ((String) st.nextToken());
        current = currentNo.trim();

        if(oldest.equals("")) && current.equals("")) {

            if(old.startsWith("+") || old.startsWith("-") ||
                old.startsWith("*") || old.startsWith("^") ||
                old.startsWith("/") || old.startsWith("==") ||
                old.startsWith(">") || old.startsWith("&") ||
                old.startsWith(""") ||
                old.startsWith("-")) {

                if(old.endsWith("+") || old.endsWith("-") ||
                    old.endsWith("*") || old.endsWith("^") ||
                    old.endsWith("/") || old.endsWith("==") ||
                    old.endsWith(">") || old.endsWith("&") ||
                    old.endsWith(""") ||
                    old.endsWith("-")) {

                    strCode = strCode+oldestNo+oldNo;
                    oldest = ""
                    oldestNo = ""
                    old = current;
                    oldNo = currentNo;
                } else {

                    //include
                    strCode = strCode + oldestNo+oldNo+currentNo;
                    oldest = ""
                    oldestNo = ""
                    old = ""
                    oldNo = ""
                }
        } else { //string

            continue;
        } else {

    //string

    } // otherwise
private String parseCommentsOut(String strLine) {
    String strCode;
    if(strLine.indexOf(Constants.COMMENT) != -1) {
        int iEndInd = strLine.indexOf(Constants.COMMENT);
        strCode = strLine.substring(0, iEndInd);
    } else {
        strCode = strLine;
    }
    return strCode;
}

private boolean funcDeclar(String strLine) {
    strLine = strLine.trim();
    int iIndex = strLine.indexOf(Constants.FUNCDECLAR);
    if(iIndex == -1) {
        return false;
    } else { 
        strLine = strLine.trim();
        //if so, and the name is not the beginning of the line
        if(iIndex != 0) {
            ...
// if the function name is not the first char of the line
// then get the preceding char
String c = strLine.substring(iIndex-1, iIndex);

// check to make sure an op,
// done to ensure name is not a substring
if(!c.equals(" ")) {
    return false;
}

// just check the last character
if((iIndex+Constants.FUNC_DECLAR.length()-1) < (strLine.length()-1)) {
    String cEnd = strLine.substring(iIndex+Constants.FUNC_DECLAR.length(),
    iIndex+Constants.FUNC_DECLAR.length()+1);
    if((cEnd.equals(" ")) || (cEnd.equals("["))) {
        return true;
    }
} else {
    // name is the last string of the line
    return true;
}
} // closes else
if(strLine.indexOf("time") != -1) {
    System.out.println("incorrect ending");
}
return false;
} // closes Function class
10.1.4 Constant

/* Constant.java
 * Quincy Scott (quincy@mit.edu)
 * Spring 2001
 * Description:
 * Represents a single constant at a single moment of time. When the name,
 * or value of the constant changes, a new object should be created.
 */

package SSPARC;

import java.util.Vector;

public class Constant extends Variable {

    public Constant(String n, String v, String u, String vr, String r, String a) {
        super(n, v, u, vr, r, a);
    }

    public Object clone() {
        Constant c = new Constant(getName(), getValue(), getUnits(),
        getValidRange(), getRationale(), getAuthor());

        c.setAliases(getAliases());

        return c;
    }
}
10.1.5 DesignVar

/* DesignVar.java
* Quincy Scott (quincy@mit.edu)
* Spring 2001
* Description:
* Represents a single design var at a single moment of time. When the
* name,
* or value of the constant changes, a new object should be created.
*/

package SSPARC;

import java.util.Vector;

public class DesignVar extends Variable {

    public DesignVar(String n, String v, String u, String vr, String r, String a) {
        super(n, v, u, vr, r, a);
    }

    public Object clone() {
        DesignVar c = new DesignVar(getName(), getValue(), getUnits(),
                                getValidRange(), getRationale(), getAuthor());
        c.setAliases(getAliases());
        return c;
    }
}

} //closes class
10.1.6 Error

package SSPARC;

public class Error extends Variable {

    public Error(String function, String msg) {
        super(function, msg, "", "", "", "");
    }

    public String getFunctionName() {
        return getName();
    }

    public String getMsg() {
        return getValue();
    }

}
10.1.7 Call

cpyackage SSPARC;

public class Call extends Variable {

    public Call(String n, String v, String u, String vr, String r,
                 String a) {
        super(n, v, u, vr, r, a);
    }

    public Object clone() {
        Call c = new Call(getName(), getValue(), getUnits(),
                          getValidRange(), getRationale(), getAuthor());

        return c;
    }

}

10.1.8 Input

/* Input.java
 * Quincy Scott (quincy@mit.edu)
 * Spring 2001
 * Description:
 * Represents a single Input.
 */

package SSPARC;

import java.util.Vector;

public class Input extends Variable {

    Vector vFields; //each entry is a InputFieldsList of the
    //InputStrucFields for this
    //input at a given time

    public Input(String n, String v, String u, String vr, String r,
        String a) {
        super(n, v, u, vr, r, a);
        vFields = new Vector();
    } //closes constructor

    public void addField(String s) {
        if(vFields.size() > 0) {
            InputFieldsList ifl = (InputFieldsList)
            vFields.lastElement();
            if(!ifl.contains(s)) {
                ifl.add(s, new InputStrucField(s, "", "", "", "", ""));
            }
        }
    }

    public void addField(InputStrucField isf) {
        if(vFields.size() > 0) {
            InputFieldsList ifl = (InputFieldsList)
            vFields.lastElement();
            if(!ifl.contains(isf.getName())) {
                ifl.add(isf.getName(), isf);
            }
        }
    }

    //called when updating the input fields at a new time
    public void updateFields() {
        vFields.addElement(new InputFieldsList());
    }
}
public InputFieldsList getCurrentFields() {
    return (InputFieldsList) vFields.lastElement();
}

public Vector getFieldsHistory() {
    return vFields;
}

public void setHistory(Vector v) {
    vFields = v;
}

public Object clone() {
    Input c = new Input(getName(), getValue(), getUnits(), getValidRange(), getRationale(), getAuthor());
    c.setAliases(getAliases());
    c.setHistory(getFieldsHistory());
    return c;
}
} //closes class
10.1.9 Output

/* Output.java
 *
 * Quincy Scott (quincy@mit.edu)
 * Spring 2001
 *
 * Description:
 * Represents a single Output.
 */

package SSPARC;

import java.util.Vector;

public class Output extends Variable {

    Vector vFields; // each entry is a OutputFieldsList of the fields
                   // for this output at a given time

    public Output(String n, String v, String u, String vr, String r, String a)
    { super(n, v, u, vr, r, a);
        vFields = new Vector();
    } // closes constructor

    public void addField(String s) {
        if(vFields.size() > 0) {
            OutputFieldsList ofl = (OutputFieldsList) vFields.lastElement();
            if(!ofl.contains(s)) {
                ofl.add(s, new OutputStrucField(s, "", "", "", "", "");
            }
        }
    }

    public void addField(OutputStrucField osf) {
        if(vFields.size() > 0) {
            OutputFieldsList ofl = (OutputFieldsList) vFields.lastElement();
            if(!ofl.contains(osf.getName())) {
                ofl.add(osf.getName(), osf);
            }
        }
    }

    // called when updating the input fields at a new time
    public void updateFields() {
        vFields.addElement(new OutputFieldsList());
    }

    public OutputFieldsList getCurrentFields() {

public Object clone() {

  Output c = new Output(getName(), getValue(), getUnits(), getValidRange(), getRationale(), getAuthor());

  c.setAliases(getAliases());
  c.setHistory(getFieldsHistory());
  return c;
}
//closes class
public class InputStrucField extends Variable {

    public InputStrucField(String n, String v, String u, String vr, String r, String a) {
        super(n, v, u, vr, r, a);
    }

    public Object clone() {
        InputStrucField c = new InputStrucField(getName(), getValue(), getUnits(), getValidRange(), getRationale(), getAuthor());
        c.setAliases(getAliases());
        return c;
    }
}
10.1.11 OutputStrucField

/* OutputStrucField.java
 * Quincy Scott (quincy@mit.edu)
 * Spring 2001
 * Description:
 * Represents a single field of an output structure at a single moment of time.
 * When the name,
 * or value of the constant changes, a new object should be created.
 */

package SSPARC;

import java.util.Vector;

public class OutputStrucField extends Variable {

    public OutputStrucField(String n, String v, String u, String vr, String r, String a) {
        super(n, v, u, vr, r, a);
    }

    public Object clone() {

        OutputStrucField c = new OutputStrucField(getName(), getValue(), getUnits(), getValidRange(), getRationale(), getAuthor());

        c.setAliases(getAliases());

        return c;
    }
}

10.1.12 FunctionList

package SSPARC;

public class FunctionList extends TimeList {

    public FunctionList() {
        super();
    }

    public Function getFunction(String s) {
        return (Function) get(s);
    }

}
10.1.13 ConstantList

package SSPARC;

public class ConstantList extends TimeList {

    public ConstantList() {
        super();
    }

    public Constant getConstant(String s) {
        return (Constant) get(s);
    }

}
package SSPARC;

public class DesignVarList extends TimeList {

    public DesignVarList() {
        super();
    }

    public DesignVar getDV(String s) {
        return (DesignVar) get(s);
    }

}
package SSPARC;
import java.util.Vector;

public class ErrorList extends TimeList {

    //each key in the hashtable is the name of a module/script/func
    //each value is a Vector of the errors for corresponding key

    public ErrorList() {
        super();
    }

    public void addError(String s, Error e) {
        Vector v;
        if(!contains(s)) {
            v = new Vector();
            v.addElement(e);
            add(s, v);
        } else {
            v = (Vector) get(s);
            v.addElement(e);
        }
    }
}


10.1.16 CallList

package SSPARC;

public class CallList extends TimeList {

    public CallList() {
        super();
    }

}

package SSPARC;

public class InputList extends TimeList {

    public InputList() {
        super();
    }

    public Input getInput(String s) {
        Object o = get(s);
        return (Input) o;
    }

}
package SSPARC;

public class OutputList extends TimeList {

    public OutputList() {
        super();
    }

    public Output getOutput(String s) {
        return (Output) get(s);
    }

}
package SSPARC;

public class InputFieldsList extends TimeList {

    public InputFieldsList() {
        super();
    }

    public InputStrucField getField(String s) {
        return (InputStrucField) get(s);
    }

} // closes class
package SSPARC;

public class OutputFieldsList extends TimeList {

    public OutputFieldsList() {
        super();
    }

    public OutputStrucField getField(String s) {
        return (OutputStrucField) get(s);
    }

}//closes class
10.1.21 Variable

/* Variable.java */
* Quincy Scott (quincy@mit.edu) * Spring 2001 *
* Description: *
* A generic class representing a variable, with a value, units, valid range, *
* rationale, author, creation date, and aliases. It is intended for classes *
* to extend this class. *
* When the name, *
* or value of the constant changes, a new object should be created. */

package SSPARC;

import java.util.Date;
import java.util.Vector;

public class Variable implements java.io.Serializable {

    private String strName;
    private String strValue;
    private String strUnits;
    private String strValidRange;
    private String strRat;
    private String strAuthor;
    private Date dDate;
    private String strAliases;

    public Variable(String n, String v, String u, String vr, String r, String a) {
        strName = n;
        strValue = v;
        strUnits = u;
        strValidRange = vr;
        strRat = r;
        strAuthor = a;
        dDate = new Date();
        strAliases = "";
    }

    public String getName() {
        return strName;
    }

    public void setName(String s) {
        strName = s;
    }

    public String getValue() {
        return strValue;
    }

}
public void setValue(String s) {
    strValue = s;
}

public String getUnits() {
    return strUnits;
}

public void setUnits(String u) {
    strUnits = u;
}

public String getValidRange() {
    return strValidRange;
}

public void setValidRange(String vr) {
    strValidRange = vr;
}

public String getRationale() {
    return strRat;
}

public void setRationale(String r) {
    strRat = r;
}

public String getAuthor() {
    return strAuthor;
}

public void setAuthor(String a) {
    strAuthor = a;
}

public Date getDate() {
    return dDate;
}

public String getAliases() {
    return strAliases;
}

public void setAliases(String s) {
    strAliases = s;
}

public boolean equals(Variable v) {
    if((!strName.equals(v.getName())) ||
       (!strValue.equals(v.getValue())) ||
       (!strUnits.equals(v.getUnits())) ||
       (!strValidRange.equals(v.getValidRange())) ||
       (!strRat.equals(v.getRationale())) ||
       (!strAuthor.equals(v.getAuthor())) ||
       (!strAliases.equals(v.getAliases())))
        return false;
    else {
        return true;
    }
}
public Variable clone() {
    Variable c = new Variable(getName(), getValue(), getUnits(),
    getValidRange(), getRationale(), getAuthor(), getDate());

    Vector v = getAliases();
    for (int i = 0; i < v.size(); i++) {
        String s = (String) v.elementAt(i);
        c.addAlias(s);
    }

    return c;
}
package SSPARC;

import java.util.Hashtable;
import java.util.Enumeration;

public class TimeList implements java.io.Serializable {

   Hashtable ht;

   public TimeList() {
      ht = new Hashtable();
   }

   public Object get(String s) {
      return (Object) ht.get(s);
   }

   public Enumeration getAll() {
      return ht.elements();
   }

   public Enumeration getAllNames() {
      return ht.keys();
   }

   public void add(String s, Variable i) {
      if(!ht.containsKey(s)) {
         ht.put(s, i);
      }
   }

   public void add(String s, Object i) {
      if(!ht.containsKey(s)) {
         ht.put(s, i);
      }
   }

   public void remove(String s) {
      if(ht.containsKey(s)) {
         ht.remove(s);
      }
   }
}

10.1.22  TimeList
public boolean contains(String s) {
    return ht.containsKey(s);
}

public int size() {
    return ht.size();
}

public Object clone() {
    TimeList il = new TimeList();
    Enumeration e = ht.keys();
    while(e.hasMoreElements()) {
        Variable v = (Variable) ht.get((String) e.nextElement());
        il.add(v.getName(), v);
    }
    return il;
}

// assumes two lists of the same size and with elements with the
// same
// name are identical. DOES NOT check all fields.
public boolean equals(TimeList tl) {
    if(!(ht.size() == tl.size())) {
        return false;
    } else {
        Enumeration eVars = tl.getAll();
        while(eVars.hasMoreElements()) {
            Variable var = (Variable) eVars.nextElement();
            if(ht.containsKey(var.getName())) {
                Variable var2 = (Variable) ht.get(var.getName());
                if(!(var.equals(var2))) {
                    return false;
                }
            }
        }
        return true;
    }
} // closes class
package SSPARC;

public class Constants {

    public final static String START_FUNC = "BTOS";

    //---------------file name constants-------------
    public final static String FILE_EXTENSION = "m";
    public final static String FILE_NAME_EXT_SEPARATOR = ".";
    public final static String DIR_SEPARATOR = "/";
    public final static String CONSTANTS_FILE_NAME = "Constants";
    public final static String DESIGN_FILE_NAME = "Design";

    //---------------code constants-----------------
    public final static String CONSTANTS_CODE = "CONSTANTS";
    public final static String DESIGN_CODE = "DESIGN";
    public final static String COMMENT = "\%";
    public final static String ASSIGN_VAL = "\=";
    public final static String ARRAY_OPEN = "\[";
    public final static String ARRAY_CLOSE = "\]";
    public final static String ARRAY_NEXT = "...";
    public final static String END_OF_LINE = "\n";
    public final static String FUNC_DECLAR = "function";
    public final static String FUNC_OPEN = "\(";
    public final static String FUNC_CLOSE = ")";
    public final static String FUNC_INPUT_SEP = ",";
    public final static String FUNC_OUTPUT_SEP = ",";
    public final static String FUNC_OUTPUT_OPEN = "\[";
    public final static String FUNC_OUTPUT_CLOSE = "]";
    public final static String STRUCT_FIELD_SEP_STR = ".";

    public final static char FUNC_OPEN_CHAR = '(';
    public final static char STRUCT_FIELD_SEP = '.';

    } //closes class
10.2 GUI

10.2.1 Btos

Written by Shane Cruz [21].

package SSPARC;

import java.awt.*;
import java.awt.event.*;

//
// Btos.java
//

public class Btos {

    private static MainFrame window = new MainFrame("SSPARC");
    private static WindowListener winListener = new WindowListener();

    public static void main(String[] args) {
        //
        // effects: Displays the main window and handle low-level events
        //

        Toolkit tk = window.getToolkit();
        Dimension wndSize = tk.getScreenSize();

        window.setBounds(25, 25, wndSize.width - 50, wndSize.height - 50);
        //window.setBounds(wndSize.width/4, wndSize.height/4, wndSize.width/2, wndSize.height/2);
window.addWindowListener(winListener);
window.setVisible(true);

static class WindowListener extends WindowAdapter {
    // Handler for window closing event
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}

}
10.2.2 MainFrame

Written by Shane Cruz [21]. Modified and extended by Quincy Scott, Tara Sainath [31], and Jason Yeung [32].

/** Mainframe.java*/

package SSPARC;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;
import java.util.*;
import javax.swing.border.*;
import java.io.File;

/** @author Shane Cruz <scruz@mit.edu>

 * Overview: Class MainFrame is an extension of JFrame.
 * It displays the main window of the application.
 *
 */

public class MainFrame extends JFrame {

    //***************************
    //** FIELDS**
    //***************************
    /* APPLICATION MENUS
    /** @specfield fileMenu : JMenu //creates a File Item in the
    Application Menu
    * @specfield projectMenu : JMenu //creates a Project Item in the
    Application Menu
    * @specfield moduleMenu : JMenu //creates a Module Item for in
    the Application Menu
    * @specfield checkMenu : JMenu //creates a Check Item in the
    Application Menu
    * @specfield helpMenu : JMenu //creates a Help Item in the
    Application Menu
    * @endspec
    */
    private JMenu fileMenu, projectMenu, moduleMenu, checkMenu, helpMenu;

    //FILE MENU ITEMS
    /** @specfield newItem : JMenuItem //creates a newProject
    Item under the File Menu
    * @specfield openItem : JMenuItem //creates a openProject
    Item under the File Menu
    * @specfield saveItem : JMenuItem //creates a saveProject
    Item under the File Menu

192
* @specfield saveProjectAsItem : JMenuItem //creates a saveProjectAs Item under the File Menu
  * @specfield closeProjectItem : JMenuItem //creates a closeProject Item under the File Menu
  * @specfield openProjectItem : JMenuItem //creates a openProject Item under the File Menu
  * @specfield exit : JMenuItem //creates a exit Item under the File Menu
  */

private JMenuItem newProjectItem, openProjectItem, saveProjectItem, saveProjectAsItem, closeProjectItem, exitItem;

    // PROJECT MENU ITEMS
    /** @specfield addFunctionItem : JMenuItem //creates an addFunction Item under the Project Menu
      * @specfield removeFunctionItem : JMenuItem //creates a removeFunction Item under the Project Menu
      * @specfield viewFunctionsItem : JMenuItem //creates a viewFunctions Item under the Project Menu
      * @specfield removeFunctionItem : JMenuItem //creates a removeFunction Item under the Project Menu
      * @specfield viewConstantsItem : JMenuItem //creates a viewConstants Item under the Project Menu
      * @specfield viewConstantsHistoryItem : JMenuItem //creates a viewConstantsHistory Item under the Project Menu
      * @specfield updateConstantsItem : JMenuItem //creates an updateConstants Item under the Project Menu
      * @specfield viewDesignVarsItem : JMenuItem //creates a viewDesignVars Item under the Project Menu
      * @specfield viewDVHistoryItem : JMenuItem //creates a viewDVHistory Item under the Project Menu
      * @specfield updateDesignVarsItem : JMenuItem //creates an updateDesignVars Item under the Project Menu
      * @specfield viewN2DiagramItem : JMenuItem //creates a viewN2Diagram Item under the Project Menu
      */

private JMenuItem addFunctionItem, removeFunctionItem, viewFunctionsItem, viewInputsItem, viewOutputsItem, viewCallsItem, viewConstantsItem, viewConstantsHistoryItem;
    private JMenuItem updateConstantsItem, viewDesignVarsItem, viewDVHistoryItem, updateDesignVarsItem, viewN2DiagramItem;

    // FUNCTION MENU ITEMS
    /** @specfield viewInputsItem : JMenuItem //creates a viewInputs Item under the Function Menu
      * @specfield viewOutputsItem : JMenuItem //creates a viewOutputs Item under the Function Menu
      * @specfield viewInputsHistoryItem : JMenuItem //creates a viewInputsHistory Item under the Function Menu
      * @specfield viewOutputsHistoryItem : JMenuItem //creates a viewOutputsHistory Item under the Function Menu
      */
* @specfield updateFunctionsItem : JMenuItem //creates a 
  updateFunctions Item under the Function Menu
* @endspec
*/
private JMenuItem viewInputsHistoryItem, viewOutputsHistoryItem,
viewCallsHistoryItem, updateFunctionsItem;

// CHECK MENU ITEMS
/** @specfield showErrorsItem : JMenuItem //creates a ShowErrors 
  Item under the Check Menu
* @endspec
*/
private JMenuItem showErrorsItem;

// HELP MENU ITEMS
/** @specfield aboutItem : JMenuItem //creates an about Item under 
  the Help Menu
* @endspec
*/
private JMenuItem aboutItem;

//CREATE PROJECT
/** @specfield currentScreen : JMenuItem // variable used to store 
  the current screen 
  * @specfield createProjButton : JButton //creates a Button called 
  "CreateProject"
  * @specfield projNameField : JTextField //creates a text field to 
  enter in the Project Name 
  * @specfield projDirField : JTextField //creates a text field to 
  enter in the Project Directory 
* @endspec
*/
public JMenuItem currentScreen;
private JButton createProjButton;
private JTextField projNameField, projDirField;

// TABLE VARIABLES
/** @specfield table : JTable //variable for displaying a table 
  * @specfield m_data : VariableTableData //variable to store the 
  variable table data 
  * @specfield : m_dataHist //variable to store the history table 
  data 
  * @specfield m_dataError : ErrorTableData //variable to store the 
  error table data 
  * @specfield ratButton : JButton //variable to store the a 
  rationale button 
  * @specfield editButton : JButton //variable to store the a edit 
  button 
* @endspec
*/
private JTable table;
private VariableTableData m_data;
private HistoryTableData m_dataHist;
private ErrorTableData m_dataError;
private JButton ratButton, editButton;
private String projectDirectory;
private MainFrame mf = this;
private String filename;
private JScrollPane windowPane;
private Container content;
private ClientIO client;
private JFileChooser files;
private RationaleDialogGUI editPane;

 SETUP

public MainFrame(String title) {
    setTitle(title);
    client = new ClientIO(); //new ClientIO object created
    filename = "untitled.ssp"; //filename of current project
        content = getContentPane();
    windowPane = new JScrollPane(content);
    currentScreen = null;
    JMenuBar menuBar = new JMenuBar();
    setJMenuBar(menuBar);
    menuBar.setBorderPainted(true);
    //Creates the text of the menu items
    fileMenu = new JMenuItem("File");
fileMenu.setMnemonic('F');
projectMenu = new JMenu("Project");
projectMenu.setMnemonic('P');
moduleMenu = new JMenu("Function");
moduleMenu.setMnemonic('M');
checkMenu = new JMenu("Check Project");
checkMenu.setMnemonic('C');
helpMenu = new JMenu("Help");
helpMenu.setMnemonic('H');

//adds the different Menu Items to the Menu Bar
menuBar.add(fileMenu);
menuBar.add(projectMenu);
menuBar.add(moduleMenu);
    menuBar.add(checkMenu);
menuBar.add(helpMenu);

//disenables the following Menu Items
projectMenu.setEnabled(false);
moduleMenu.setEnabled(false);
checkMenu.setEnabled(false);

//new instance of KeyMenuAction is created
KeyMenuAction aListener = new KeyMenuAction();

//***************************************************************************
//Code for the FILE menu
//***************************************************************************

//ADDS ITEMS INTO THE FILE MENU
//(NewProject, OpenProject, SaveProject, SaveProjectAs, CloseProject, Exit)

fileMenu.add(newProjectItem = new JMenuItem("New Project", 'N'));
//newProjectItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N,
Event.CTRL_MASK, false));
    newProjectItem.addActionListener(aListener);

fileMenu.add(openProjectItem = new JMenuItem("Open Project...", 'O'));
//openProjectItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,
Event.CTRL_MASK, false));
    openProjectItem.addActionListener(aListener);

fileMenu.add(saveProjectItem = new JMenuItem("Save Project", 'S'));
//saveProjectItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S,
Event.CTRL_MASK, false));
    saveProjectItem.addActionListener(aListener);

fileMenu.add(saveProjectAsItem = new JMenuItem("Save Project As...", 'A'));

196
//saveProjectAsItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A, Event.CTRL_MASK, false));
saveProjectAsItem.addActionListener(aListener);

fileMenu.add(closeProjectItem = new JMenuItem("Close Project", 'C'));

//saveProjectAsItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A, Event.CTRL_MASK, false));
closeProjectItem.addActionListener(aListener);

fileMenu.addSeparator();

fileMenu.add(exitItem = new JMenuItem("Exit", 'X'));
//exit Item. setAccelerator (KeyStroke. getKeyStroke (KeyEvent.VKX, Event.CTRLMASK, false));
exitItem.addActionListener(aListener);

openProjectItem.setEnabled(false);
saveProjectItem.setEnabled(false);
saveProjectAsItem.setEnabled(false);

/*************************************************************/
//Code for the Project menu
/*************************************************************/

//ADDS ITEMS INTO THE PROJECT MENU
//(Add Function, Remove Function, View Functions, View Constants, Constants History, View DV, DV History, Update DV)

projectMenu.add(addFunctionItem = new JMenuItem("Add Function", 'A'));

//addFunctionItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A, 3, false));
addFunctionItem.addActionListener(aListener);

projectMenu.add(removeFunctionItem = new JMenuItem("Remove Function", 'R'));

//removeFunctionItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_R, 3, false));
removeFunctionItem.addActionListener(aListener);

projectMenu.add(viewFunctionsItem = new JMenuItem("View a List of All Functions", 'M'));

//viewFunctionsItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_V, 3, false));
viewFunctionsItem.addActionListener(aListener);

moduleMenu.add(viewInputsItem = new JMenuItem("View Function Inputs", 'V'));

197
//viewInputsItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_V, 2, false));
viewInputsItem.addActionListener(aListener);
moduleMenu.add(viewOutputsItem = new JMenuItem("View Function Outputs", 'O'));

//viewOutputsItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O, 2, false));
viewOutputsItem.addActionListener(aListener);
moduleMenu.add(viewCallsItem = new JMenuItem("View Function Calls", 'Z'));
viewCallsItem.addActionListener(aListener);
projectMenu.addSeparator();
projectMenu.add(viewConstantsItem = new JMenuItem("View Constants", 'C'));
//viewConstantsItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_V, 2, false));
viewConstantsItem.addActionListener(aListener);
projectMenu.add(viewConstantsHistoryItem = new JMenuItem("View History of Constants", 'H'));
//viewConstantsHistoryItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_V, 2, false));
viewConstantsHistoryItem.addActionListener(aListener);
projectMenu.add(updateConstantsItem = new JMenuItem("Update Constants", 'U'));
//updateConstantsItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_U, 2, false));
updateConstantsItem.addActionListener(aListener);
projectMenu.addSeparator();
projectMenu.add(viewDesignVarsItem = new JMenuItem("View Design Variables", 'D'));
//viewDesignVarsItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_D, 2, false));
viewDesignVarsItem.addActionListener(aListener);
projectMenu.add(viewDVHistoryItem = new JMenuItem("View History of Design Variables", 'I'));
//viewDVHistoryItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A, 3, false));
viewDVHistoryItem.addActionListener(aListener);
projectMenu.add(updateDesignVarsItem = new JMenuItem("Update Design Variables", 'P'));

//updateDesignVarsItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_U, 2, false));
updateDesignVarsItem.addActionListener(aListener);

//Tara's addition
projectMenu.addSeparator();

projectMenu.add(viewN2DiagramItem = new JMenuItem("View N^2 Diagram", 'N'));

//viewN2Diagram.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N, 2, false));
viewN2DiagramItem.addActionListener(aListener);

/***********************
// Code for the Function menu
 ********************/
moduleMenu.add(viewInputsHistoryItem = new JMenuItem("View History of Inputs", 'I'));

//viewInputsHistoryItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_I, 2, false));
viewInputsHistoryItem.addActionListener(aListener);

moduleMenu.add(viewOutputsHistoryItem = new JMenuItem("View History of Outputs", 'P'));

//viewOutputsHistoryItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_P, 2, false));
viewOutputsHistoryItem.addActionListener(aListener);

moduleMenu.add(viewCallsHistoryItem = new JMenuItem("View History of Calls", 'Y'));
viewCallsHistoryItem.addActionListener(aListener);

moduleMenu.add(updateFunctionsItem = new JMenuItem("Update all Functions", 'U'));

//updateFunctionsItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_U, 2, false));
updateFunctionsItem.addActionListener(aListener);

/***********************
// Code for the Check Project menu
 **********************/
//ADDS ITEMS TO THE CHECK PROJECT MENU
//(Show Errors)

checkMenu.add(showErrorsItem = new JMenuItem("Show Errors", 'S'));

//aboutItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A, 2, false));
showErrorsItem.addActionListener(aListener);

/*******************************************************************************
// Code for the Help menu
*******************************************************************************/

//ADDS ITEMS TO THE HELP MENU
//(About)

helpMenu.add(aboutItem = new JMenuItem("About", 'A'));
aboutItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A, 2, false));
aboutItem.addActionListener(aListener);

/*******************************************************************************
projectClosed();
files = new JFileChooser(".");
} //closes constructor MainFrame

// --------------
// PRIVATE CLASS
// --------------

/*** Overview: Class KeyMenuAction implements ActionListener
 * It highlights the appropriate action that is performed
 * when a specific event occurs
 */
class KeyMenuAction implements ActionListener {

public void actionPerformed(ActionEvent event) {

//variable object bound to source of event
Object object = event.getSource();

// FILE MENU ITEMS

//if the object is a new Project Item
if (object == newProjectItem) {
    content.removeAll();
    JLabel projNameLabel = new JLabel("Enter Project Name: ");
    getContentPane().add(projNameLabel);
}
projNameLabel.setForeground(Color.black);
projNameLabel.setBounds(50, 50, 200, 15);

projNameField = new JTextField(25);
getContentPane().add(projNameField);
projNameField.setBounds(175, 40, 200, 30);

JLabel projDirLabel = new JLabel("Enter Project Directory: ");
getContentPane().add(projDirLabel);
projDirLabel.setForeground(Color.black);
projDirLabel.setBounds(30, 115, 200, 15);

projDirField = new JTextField(50);
getContentPane().add(projDirField);
projDirField.setBounds(175, 105, 300, 30);

createProjButton = new JButton("Create Project");
getContentPane().add(createProjButton);
createProjButton.setBounds(175, 165, 150, 30);

projNameField.requestFocus();
projNameField.addActionListener(this);
projDirField.addActionListener(this);
createProjButton.addActionListener(this);
repaint();

if (object == createProjButton) {
    String projName = projNameField.getText();
    String projDir = projDirField.getText();

    java.io.File f = new File(projDir);

    if (projName.equals("")) {
        JOptionPane.showMessageDialog(mf, "You must enter a project name!", "Error", JOptionPane.ERROR_MESSAGE);
    } else if (!f.isDirectory()) {
        JOptionPane.showMessageDialog(mf, "That directory does not exist!", "Error", JOptionPane.ERROR_MESSAGE);
    } else {
        try {
            client.createProject(projName, projDir);
            setTitle("SSPARC - " + projName.toUpperCase());
            projectDirectory = projDir;
            content.removeAll();
            projectOpen();
            filename = projName.toLowerCase() + ".ssp";
            files = new JFileChooser(projectDirectory);
            repaint();
        } catch (Exception ex) {
            System.out.println(ex);
            JOptionPane.showMessageDialog(mf, "Unable to create new project: " + ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
} // closes object =

if (object == projNameField) {
    projDirField.requestFocus();
}
if (object == projDirField) {
    createProjButton.doClick();
}
if (object == openProjectItem) {
    MyFileFilter filter = new MyFileFilter("ssp", "SSPARC Projects (.ssp)");
    files.setFileFilter(filter);
    File file = showDialog("Open Project", "Open", "Open a Project", 'o', null);
    if (file != null) {
        String fname = file.getPath() + "/" + file.getName();
        filename = fname;
        int key = fname.indexOf(".ssp");
        if (key == -1) {
            JOptionPane.showMessageDialog(mf, "Unable to Open Project: That is not a valid SSPARC file", "Btos Organizer - Error Message", JOptionPane.ERROR_MESSAGE);
        } else {
            fname = fname.substring(0, key);
            try {
                client.openProject(fname);
                projectOpen();
                setTitle("SSPARC - " + client.getProjectName().toUpperCase());
                content.removeAll();
                repaint();
            } catch (Exception e) {
                JOptionPane.showMessageDialog(mf, "Unable to Open Project: " + e.getMessage(), "Btos Organizer - Error Message", JOptionPane.ERROR_MESSAGE);
            }
        }
    }
}
if (object == saveProjectItem) {
    saveProjectAsItem.doClick();
}
if (object == saveProjectAsItem) {
    files.setFileFilter(new MyFileFilter("ssp", "SSPARC Projects (.ssp)"));
    File file = showDialog("Save Project As...", "Save", "Save the Project", 's', new File(filename));
if (file != null) {
    filename = file.getPath() + "/" + file.getName();
    int key = filename.indexOf(".ssp");
    filename = filename.substring(0, key);
    try {
        client.saveProject(filename);
    } catch (Exception e) {
        System.err.println("Error: " + e.getMessage());
    }
}

if (object == closeProjectItem) {
    //client = new ClientIO();
    content.removeAll();
    projectClosed();
    setTitle("SSPARC");
    filename = null;
    repaint();
}

if (object == exitItem) {
    System.exit(0);
}

// Project Menu
if (object == addFunctionItem) {
    //showFunctions();

    files.setFileFilter(new MyFileFilter("m", "MATLAB files (.m)");

    File file = showDialog("Add a Function", "Add", "Add a Function", 'a', null);

    if (file != null) {
        String fname = file.getName();
        String fullName = fname;
        int key = fname.indexOf(".m");

        fname = fname.substring(0, key);

        try { 
            client.addFunction(fullName);
            JOptionPane.showMessageDialog(mf, "Function " + fname + ".m added to the project", "Btos Organizer", 
            JOptionPane.INFORMATION_MESSAGE);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(mf, "Unable to Add Function: " + e.getMessage(), "Btos Organizer - Error Message", 
            JOptionPane.ERROR_MESSAGE);
        }
    }
}

if (object == removeFunctionItem) {

String modName = selectFunction();
if (modName != null) {
    try {
        client.removeFunction(modName);
    } catch (Exception e) {
        JOptionPane.showMessageDialog(mf, "Unable to remove module", "Btos Organizer - Error", JOptionPane.ERROR_MESSAGE);
    }
}
if (currentScreen != null) currentScreen.doClick();
//showFunctions();

if (object == viewFunctionsItem) {
    showVariableTable(client.getFunctions(), "Functions");
    currentScreen = viewFunctionsItem;
}

if (object == viewInputsItem) {
    showInputTable(client.getFunctions(), "Function Inputs");
    currentScreen = viewInputsItem;
}
if (object == viewOutputsItem) {
    showOutputTable(client.getFunctions(), "Function Outputs");
    currentScreen = viewOutputsItem;
}
if (object == viewCallsItem) {
    showCallsTable(client.getFunctions(), "Function Calls");
    currentScreen = viewCallsItem;
}
if (object == viewConstantsItem) {
    showVariableTable(client.getConstants(), "Constants");
    currentScreen = viewConstantsItem;
}
if (object == viewConstantsHistoryItem) {
    //showHistoryTable(createConstantListVector());
    showHistoryTable(client.getConstantsHist(), "Constants History");
    currentScreen = viewConstantsHistoryItem;
}
if (object == updateConstantsItem) {
    try {
        client.updateConstants();
    }
}
JOptionPane.showMessageDialog(mf, "Constants Successfully Updated", "Btos Organizer - Constants Update", JOptionPane.INFORMATION_MESSAGE);
} catch (java.io.IOException e) {
    JOptionPane.showMessageDialog(mf, "Constants Unable to be Updated: " + e.getMessage(), "Btos Organizer - Constants Update", JOptionPane.WARNING_MESSAGE);
}
if (currentScreen != null) currentScreen.doClick();
}

if (object == viewDesignVarsItem) {
    currentScreen = viewDesignVarsItem;
    showVariableTable(client.getDesignVars(), "Design Variables");
}

if (object == viewDVHistoryItem) {
    showHistoryTable(client.getDesignVarsHist(), "Design Variables History");
    currentScreen = viewDVHistoryItem;
}

if (object == updateDesignVarsItem) {
    try {
        client.updateDesignVars();
        JOptionPane.showMessageDialog(mf, "Design Variables Successfully Updated", "Btos Organizer - DV Update", JOptionPane.INFORMATION_MESSAGE);
    } catch (java.io.IOException e) {
        JOptionPane.showMessageDialog(mf, "Design Variables Unable to be Updated: " + e.getMessage(), "Btos Organizer - DV Update", JOptionPane.WARNING_MESSAGE);
    }
    if (currentScreen != null) currentScreen.doClick();
}

// Tara's additions
if (object == viewN2DiagramItem) {
    Enumeration eFunctions = client.getFunctionsNames();
    Vector vFunctions = new Vector();
    while (eFunctions.hasMoreElements()){
        vFunctions.add(eFunctions.nextElement());
    }
    SsparcTable app = new SsparcTable(vFunctions, client);
    app.show();
    currentScreen = viewN2DiagramItem;
}

if (object == showErrorsItem) {
    ErrorList eList = null;
    try {
        eList = client.checkProject();
        currentScreen = showErrorsItem;
    } catch (Exception e) {
JOptionPane.showMessageDialog(mf, "Unable to Check the Project: " + e.getMessage(), "Btos Organizer - Check Project", JOptionPane.WARNING_MESSAGE);

showErrorsTable(eList);

// Function Menu
// Jason's changes

if (object == viewOutputsItem) {
    showVariableTable(client.getOutput(), "Function Outputs");
    currentScreen = viewOutputsItem;
}

if (object == viewInputsHistoryItem) {
    String strName = selectFunction();
    Function f = client.getFunction(strName);
    Vector vYes = new Vector();
    Vector vIns = f.getInputsHist();
    for(int i=0; i<vIns.size(); i++) {
        InputList il = (InputList) vIns.elementAt(i);
        Enumeration e = il.getAllNames();
        String ins = " ";
        while(e.hasMoreElements()) {
            ins += (String) e.nextElement() + " ";
        }
        TimeList tl = new TimeList();
        tl.add(f.getName(), new FuncIO(f.getName(),
            ins.trim()));
        vYes.addElement(tl);
    }
    showIOHistoryTable(vYes, "Inputs History");
    currentScreen = viewInputsHistoryItem;
} //closes viewInputsHistoryItem

if (object == viewOutputsHistoryItem) {
    String strName = selectFunction();
    Function f = client.getFunction(strName);
    Vector vYes = new Vector();
    Vector vIns = f.getOutputsHist();
    for(int i=0; i<vIns.size(); i++) {
        OutputList il = (OutputList) vIns.elementAt(i);
Enumeration e = il.getAllNames();

String ins = " ";
while (e.hasMoreElements()) {
    ins += (String) e.nextElement() + " ";
}

TimeList tl = new TimeList();
tl.add(f.getName(), new FuncIO(f.getName(),
ins.trim()));

vYes.addElement(tl);
}

showIOHistoryTable(vYes, "Outputs History");
currentScreen = viewOutputsHistoryItem;

if (object == viewCallsHistoryItem) {
    String strName = selectFunction();
    Function f = client.getFunction(strName);
    Vector vYes = new Vector();
    Vector vIns = f.getCalledHist();
    for (int i = 0; i < vIns.size(); i++) {
        CallList il = (CallList) vIns.elementAt(i);
        Enumeration e = il.getAllNames();

        String ins = " ";
        while (e.hasMoreElements()) {
            ins += (String) e.nextElement() + " ";
        }

        TimeList tl = new TimeList();
tl.add(f.getName(), new FuncIO(f.getName(),
        ins.trim()));

        vYes.addElement(tl);
    }

    showIOHistoryTable(vYes, "Calls History");
currentScreen = viewCallsHistoryItem;
}

if (object == updateFunctionsItem) {
    try {
        client.updateAll();
        JOptionPane.showMessageDialog(mf, "Functions Successfully Updated", "Btos Organizer - Constants Update", JOptionPane.INFORMATION_MESSAGE);
    } catch (java.io.IOException e) {
        
207
JOptionPane.showMessageDialog(mf, "Functions Unable to be Updated: " + e.getMessage(), "Btos Organizer - Constants Update", JOptionPane.WARNING_MESSAGE);

} catch (java.lang.Exception e)
{
    JOptionPane.showMessageDialog(mf, "Fuck u" + e.getMessage(), "FUck u too", JOptionPane.WARNINGMESSAGE);

}

if (currentScreen != null) currentScreen.doClick();

// Help Menu
if (object == aboutItem) {
    JOptionPane.showMessageDialog(mf, "About Btos Organizer", "Btos Organizer - Version .0000001", JOptionPane.INFORMATIONMESSAGE);
}

/
} //closes class KeyActionMenu

// ---------------------
// PRIVATE METHODS
// ---------------------

/** @effects allows the user to enter a rationale for a certain function */

public String getUserModRat(String modName) {
    String rationale = "";
    if (modName != null) {
        //allows the user to enter rationale for a certain function
        rationale = JOptionPane.showInputDialog(mf, "Enter the module rationale for " + modName + ".m: ", "Function Rationale", JOptionPane.QUESTIONMESSAGE);
    }
    return rationale; //rationale is returned
}

/** @effects allows the user to enter a rationale for a Design Variable */

public String getUserDVRat(String dvName) {
    String rationale = "";
    if (dvName != null) {
        //allows the user to enter rationale for a Design Variable
        rationale = JOptionPane.showInputDialog(mf, "Enter the design variable rationale for " + dvName + ": ", "Design Variable Rationale", JOptionPane.QUESTIONMESSAGE);
    }
    return rationale; //rationale is returned
}

/** @requires String s !=null */
* @effects creates a Title of String s and returns it
* /

private String createTitle(String s) {
    //return "<HTML><B><FONT SIZE=+1>" + s + "</FONT></B></HTML>";
    return s;
}

/** @requires Enumeration e, String title !=null
   * @modifies (Container content) by adding to it a border, topPanel and MainPanel
   * @effects a scrollable table-view with borders is added to the content
   * Also, the table associated with a certain variable is displayed.
   */

private void showVariableTable(Enumeration e, String title) {

    //vars holds a list of all the enumerations e
    Enumeration vars = e;

    //instanciates new vectors, bound to data and variables
    Vector data = new Vector();
    Vector variables = new Vector();

    //while-loop adds all elements of vars to vector variable
    //and specific info. about vars to the vector data (as discussed below)
    while (vars.hasMoreElements()) {
        Variable v = (Variable) vars.nextElement();
        variables.addElement(v);
        data.addElement(new VariableData(v.getName(), formatForTable(v.getValue()), v.getUnits(), v.getValidRange(), v.getAuthor(), v.getDate(), v.getAliases())); //add a new variable to data with a name, value, units, etc.
    }

    //remove what was previously on the Screen
    content.removeAll();

    //sets the layout of content to have a border
    BorderLayout border = new BorderLayout();
    content.setLayout(border);

    // Set the horizontal gap between components
    border.setHgap(20);

    //creates a new border of type BevelBorder
    BevelBorder edge = new BevelBorder(BevelBorder.RAISED);

    //creates two panels
    JPanel topPanel, mainPanel;

    //Add topPanel into content
    //The topPanel contains the data, variables and title
content.add(topPanel = createTopPanel(data, variables, title), BorderLayout.NORTH);
    //Add the main Panel to content
    content.add(mainPanel = new JPanel(), BorderLayout.CENTER);

    //create the border for the panels
topPanel.setBorder(edge);
    mainPanel.setBorder(edge);

    //Create the scroll pane and add the table to it.
JScrollPane scrollPane = new JScrollPane(table);

    //Add the scroll pane to this window (which in effects adds the
    //table to the window)
mainPanel.add scrollPane, BorderLayout.CENTER);
    scrollPane.setVisible(true);
    repaint();

private void showInputTable(Enumeration e, String title) {

    //vars holds a list of all the enumerations e
    Enumeration vars = e;

    //instanciates new vectors, bound to data and variables
    Vector data = new Vector();
    Vector variables = new Vector();

    //while-loop adds all elements of vars to vector variable
    //and specific info. about vars to the vector data (as discussed
    //below)
while (vars.hasMoreElements()) {
        Function v = (Function) vars.nextElement();
        variables.addElement(v);
        String ins = " ";
        Enumeration inName = v.getCurrentInputs().getAllNames();

        while (inName.hasMoreElements()) {
            ins += (String) inName.nextElement() + " ";
        }

        data.addElement(new VariableData(v.getName(),
                                    formatForTable(ins.trim()),
                                    v.getUnits(),
                                    v.getValidRange(),
                                    v.getAuthor(),
                                    v.getDate(),
                                    v.getAliases());

    }
} //closes method showVariableTable
//remove what was previously on the Screen
content.removeAll();

//sets the layout of content to have a border
BorderLayout border = new BorderLayout();
content.setLayout(border);

// Set the horizontal gap between components
border.setHgap(20);

//creates a new border of type BevelBorder
BevelBorder edge = new BevelBorder(BevelBorder.RAISED);

//creates two panels
    JPanel topPanel, mainPanel;

//Add topPanel into content
//The topPanel contains the data, variables and title
content.add(topPanel = createTopPanel(data, variables, title), BorderLayout.NORTH);
//Add the main Panel to content
content.add(mainPanel = new JPanel(), BorderLayout.CENTER);

//create the border for the panels
topPanel.setBorder(edge);
mainPanel.setBorder(edge);

//Create the scroll pane and add the table to it.
JScrollPane scrollPane = new JScrollPane(table);

//Add the scroll pane to this window (which in effects adds the
table to the window)
mainPanel.add(scrollPane, BorderLayout.CENTER);
scrollPane.setVisible(true);
repaint();
} //closes method showInputTable

private void showOutputTable(Enumeration e, String title) {

    //vars holds a list of all the enumerations e
    Enumeration vars = e;

    //instanciates new vectors, bound to data and variables
    Vector data = new Vector();
    Vector variables = new Vector();

    //while-loop adds all elements of vars to vector variable
    //and specific info. about vars to the vector data (as discussed
    //below)
    while (vars.hasMoreElements()) {
        Function v = (Function) vars.nextElement();
        variables.addElement(v);
        String ins = " ";

        String ins = " ";

} //closes method showOutputTable
Enumeration inName = v.getCurrentOutputs().getAllNames();

while (inName.hasMoreElements()) {
    ins += (String) inName.nextElement() + " ";
}

data.addElement(new VariableData(v.getName(), formatForTable(ins.trim()), v.getUnits(), v.getValidRange(), v.getAuthor(), v.getDate(), v.getAliases())); // add a new variable to data with a name, value, units, etc.

    // data.addElement(new VariableData(v.getName(), ins.trim(), outs.trim(), ", "", "", v.getDate(), ")
    }

// remove what was previously on the Screen
content.removeAll();

// sets the layout of content to have a border
BorderLayout border = new BorderLayout();
content.setLayout(border);

// Set the horizontal gap between components
border.setHgap(20);

// creates a new border of type BevelBorder
BevelBorder edge = new BevelBorder(BevelBorder.RAISED);

// creates two panels
JPanel topPanel, mainPanel;

// Add topPanel into content
// The topPanel contains the data, variables and title
content.add(topPanel = createTopPanel(data, variables, title), BorderLayout.NORTH);
// Add the main Panel to content
content.add(mainPanel = new JPanel(), BorderLayout.CENTER);

// create the border for the panels
topPanel.setBorder(edge);
mainPanel.setBorder(edge);

// Create the scroll pane and add the table to it.
JScrollPane scrollPane = new JScrollPane(table);

// Add the scroll pane to this window (which in effects adds the table to the window)
mainPanel.add(scrollPane, BorderLayout.CENTER);
scrollPane.setVisible(true);
repaint();
} // closes method showOutputTable
private void showCallsTable(Enumeration e, String title) {
    // vars holds a list of all the enumerations e
    Enumeration vars = e;

    // instantiates new vectors, bound to data and variables
    Vector data = new Vector();
    Vector variables = new Vector();

    // while-loop adds all elements of vars to vector variable
    // and specific info. about vars to the vector data (as discussed below)
    while (vars.hasMoreElements()) {
        Function v = (Function) vars.nextElement();
        variables.addElement(v);
        String ins = " ";
        Enumeration inName = v.getCurrentCalled().getAllNames();
        while (inName.hasMoreElements()) {
            ins += (String) inName.nextElement() + " ";
        }
        data.addElement(new VariableData(v.getName(),
            formatForTable(ins.trim(), v.getUnits(), v.getValidRange(),
            v.getAuthor(), v.getDate(), v.getAllases())); // add a new variable to
data with a name, value, units, etc.
        //
        //data.addElement(new VariableData(v.getName(),
        //    ins.trim(), outs.trim(), "", "", v.getDate(), ");
    }

    // remove what was previously on the Screen
    content.removeAll();

    // sets the layout of content to have a border
    BorderLayout border = new BorderLayout();
    content.setLayout(border);

    // Set the horizontal gap between components
    border.setHgap(20);

    // creates a new border of type BevelBorder
    BevelBorder edge = new BevelBorder(BevelBorder.RAISED);

    // creates two panels
    JPanel topPanel, mainPanel;

    // Add topPanel into content
    // The topPanel contains the data, variables and title
    content.add(topPanel = createTopPanel(data, variables, title),
        BorderLayout.NORTH);
}
//Add the main Panel to content
content.add(mainPanel = new JPanel(), BorderLayout.CENTER);

//create the border for the panels
topPanel.setBorder(edge);
mainPanel.set Border(edge);

//Create the scroll pane and add the table to it.
JSchorllPane scrollPane = new JScrollPane(table);

//Add the scroll pane to this window (which in effect adds the
table to the window)
mainPanel.add(scrollPane, BorderLayout.CENTER);
scrollPane.setVisible(true);
repaint();
} //closes method showCallTable

/** @requires Vector v, String title !=null
 * @modifies (Container content) by adding a border, panels and
 * table
 * @modifies (JTable table) by adding certain view properties and
 * columns with variable data
 * @modifies (HistoryTableData m_dataHist) by adding variable data
 * @effects establishes a table with the history of all variables
 * created
 */

private void showHistoryTable(Vector v, String title)
{

    //three new vector's are created
    Vector data = new Vector();
    Vector varList = new Vector();
    Vector varNameList = new Vector();

    //loop over the size of the Vector v
    for(int i=0; i < v.size(); i++) {

        //get element of v at spot i and add it to timelist
        Timel i = (TimeList) v.get(i);
        //add to vars all elements of timelist
        Enumeration vars = t.getAll();

        //loop over the number of elements in vars
        while (vars.hasMoreElements()) {
            Variable curVar = (Variable) vars.nextElement();
            //if the variable is not contained in the VarNameList,
            //add the variable to the varList and
            //it's name to the varNameList
            if (!varNameList.contains(curVar.getName())) {
                varList.addElement(curVar);
                varNameList.addElement(curVar.getName());
            }
        } //closes while-loop
    } //closes for-loop

    //to summarize: VarList contains a list of all the variables
//VarNameList contains a list of the variables' names

//loop over the size of the varList
for(int i=0; i < varList.size(); i++) {
    //add each element of varList to Vector data
    Variable curVar = (Variable) varList.get(i);
    data.addElement(new HistoryData(curVar, v));
} //closes for-loop

content.removeAll();
//sets the layout of content to have a border
BorderLayout border = new BorderLayout();
content.setLayout(border);

// Set the horizontal gap between components
border.setHgap(20);

//creates a new border of type BevelBorder
BevelBorder edge = new BevelBorder(BevelBorder.RAISED);

//creates 4 new Panels
JPanel topPanel, mainPanel, titlePanel, keyPanel;

//makes a rectangular grid of 2 rows and 1 column
topPanel = new JPanel(new GridLayout(2,1));

//add the topPanel to the content
content.add(topPanel, BorderLayout.NORTH);

//add the titlePanel and keyPanel to the content where the layout will be center
   topPanel.add(titlePanel = new JPanel(new FlowLayout(FlowLayout.CENTER)));
topPanel.add(keyPanel = new JPanel(new FlowLayout(FlowLayout.CENTER)));

   //add a title to the titlePanel with font SanSerif
   JLabel titleLabel;
titlePanel.add(titleLabel = new JLabel(createTitle(title)));
titleLabel.setFont(new Font("SanSerif", Font.BOLD, 18));

   //add label "Color Key: " to the key Panel
   keyPanel.add(new JLabel("Color Key: "));

   //creates three new buttons
   JButton greenButton, grayButton, redButton;

   //establishes that Green means a variable has been added
   keyPanel.add(greenButton = new JButton());
greenButton.setBackground(Color.green);
greenButton.setEnabled(false);
   keyPanel.add(new JLabel("Variable Added "));

   //establishes that Gray means a Variable has been removed
   keyPanel.add(grayButton = new JButton());
grayButton.setBackground(Color.gray);
grayButton.setEnabled(false);
keyPanel.add(new JLabel("Variable Removed 
//establishes that Red means a variable has been changed
keyPanel.add(redButton
redButton.setBackground(Color.red); redButton.setEnabled(false);
keyPanel.add(new JLabel("Variable Changed 

//a new history table has been created that shows the data of the variables
m_dataHist = new HistoryTableData(data, v);

//make a new table
table = new JTable();

//below preferences of the table are established:

//set's table selection at a single selection
table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
//set size of viewport of table
table.setPreferredScrollableViewportSize(new Dimension(content.getWidth()-50, content.getHeight()-150));
//establishes that the table cannot be resized
table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);

//cannot create columns by self
table.setAutoCreateColumnsFromModel(false);
//sets the data for the table to be m_dataHist (which contains all the variables)
table.setModel(m_dataHist);

//lop over the size of the variables again
for (int k=0; k <= v.size(); k++) {
    //displays an individual cell in a table
    DefaultTableCellRenderer renderer = new ColoredTableCellRenderer();
    //the text of the cell is aligned left
    renderer.setHorizontalAlignment(JLabel.LEFT);
    //create a new column with index at k, width at 200, the cell is rendered and cannot be edited
    //basically, a column is created with index at k.
    TableColumn column = new TableColumn(k, 200, renderer, null);
    column.setMinWidth(200);
    table.addColumn(column); //add the column to the table
}

//get table-header and set it's properties
JTableHeader header = table.getTableHeader();
header.setUpdateTableInRealTime(false);
header.setReorderingAllowed(false);

//add the mainPanel to content
content.add(mainPanel = new JPanel(), BorderLayout.CENTER);

//establish the border of the mainPanel
mainPanel.setBorder(edge);

//Create the scroll pane and add the table to it.
JScrollPane scrollPane = new JScrollPane(table);

//Add the scroll pane to this window.
mainPanel.add(scrollPane, BorderLayout.CENTER);
scrollPane.setVisible(true);
repaint();
}  //closes showHistoryTable method

private void showIOHistoryTable(Vector v, String title) {

    //three new vector's are created
    Vector data = new Vector();
    Vector varList = new Vector();
    Vector varNameList = new Vector();

    //loop over the size of the Vector v
    for(int i=0; i < v.size(); i++) {

        //get element of v at spot i and add it to timelist
        TimeList t = (TimeList) v.get(i);
        //add to vars all elements of timelist
        Enumeration vars = t.getAll();

        //loop over the number of elements in vars
        while (vars.hasMoreElements()) {
            FuncIO curVar = (FuncIO) vars.nextElement();
            //if the variable is not contained in the VarNameList,
            //add the variable to the varList and
            //it's name to the varNameList
            if (!varNameList.contains(curVar.getName())) {
                varList.addElement(curVar);
                varNameList.addElement(curVar.getName());
            }
        }
    }

    //to summarize: VarList contains a list of all the variables
    //VarNameList contains a list of the varables' names

    //loop over the size of the varList
    for(int i=0; i < varList.size(); i++) {
        //add each element of varList to Vector data
        Variable var = (Variable) varList.get(i);
        //
        data.addElement(new HistoryData(var, v));
    }

    content.removeAll();
    //sets the layout of content to have a border
    BorderLayout border = new BorderLayout();
    content.setLayout(border);
}
// Set the horizontal gap between components
border.setHgap(20);

// creates a new border of type BevelBorder
BevelBorder edge = new BevelBorder(BevelBorder.RAISED);

// creates 4 new Panels
JPanel topPanel, mainPanel, titlePanel, keyPanel;

// makes a rectangular grid of 2 rows and 1 column
topPanel = new JPanel(new GridLayout(2, 1));

// add the topPanel to the content
content.add(topPanel, BorderLayout.NORTH);

// add the titlePanel and keyPanel to the content where the layout will be center
  topPanel.add(titlePanel = new JPanel(new FlowLayout(FlowLayout.CENTER)));
  topPanel.add(keyPanel = new JPanel(new FlowLayout(FlowLayout.CENTER)));

// add a title to the titlePanel with font SanSefir
JLabel titleLabel;
titlePanel.add(titleLabel = new JLabel(createTitle(title)));
titleLabel.setFont(new Font("SanSerif", Font.BOLD, 18));

// add label "Color Key: " to the key Panel
keyPanel.add(new JLabel("Color Key: "));

// creates three new buttons
JButton greenButton, grayButton, redButton;

// establishes that Green means a variable has been added
keyPanel.add(greenButton = new JButton());
greenButton.setBackground(Color.green);
greenButton.setEnabled(false);
keyPanel.add(new JLabel("Variable Added "));

// establishes that Gray means a variable has been removed
keyPanel.add(grayButton = new JButton());
grayButton.setBackground(Color.gray);
grayButton.setEnabled(false);
keyPanel.add(new JLabel("Variable Removed "));

// establishes that Red means a variable has been changed
keyPanel.add(redButton = new JButton());
redButton.setBackground(Color.red);
redButton.setEnabled(false);
keyPanel.add(new JLabel("Variable Changed "));

// a new history table has been created that shows the data of the variables
m_dataHist = new HistoryTableData(data, v);
// make a new table
table = new JTable();

// below preferences of the table are established:

// set's table selection at a single selection
setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

// set size of viewport of table
setPreferredSize(new Dimension(content.getWidth() - 50, content.getHeight() - 150));

// establishes that the table cannot be resized
setAutoResizeMode(JTable.AUTO_RESIZE_OFF);

// cannot create columns by self
setAutoCreateColumnsFromModel(false);

// sets the data for the table to be m_dataHist (which contains all the variables)
setModel(m_dataHist);

// loop over the size of the variables again
for (int k = 0; k <= v.size(); k++) {
    // displays an individual cell in a table
    DefaultTableCellRenderer renderer = new ColoredTableCellRenderer();
    // the text of the cell is aligned left
    renderer.setHorizontalAlignment(JLabel.LEFT);
    // create a new column with index at k, width at 200, the cell is rendered and cannot be edited
    // basically, a column is created with index at k.
    TableColumn column = new TableColumn(k, 200, renderer, null);
    column.setMinWidth(200);
    table.addColumn(column); // add the column to the table
}

// get table-header and set it's properties
getTableHeader();
header.setUpdateTableInRealTime(false);
header.setReorderingAllowed(false);

// add the mainPanel to content
add(mainPanel = new JPanel(), BorderLayout.CENTER);

// establish the border of the mainPanel
mainPanel.setBorder(edge);

// create the scroll pane and add the table to it.
ScrollPane scrollPane = new JScrollPane(table);

// add the scroll pane to this window.
mainPanel.add(scrollPane, BorderLayout.CENTER);
scrollPane.setVisible(true);
repaint();
} // closes showInputHistoryTable method

/** @requires ErrorList eList !=null */
private void showErrorsTable(ErrorList eList) {

//get all the names from the ErrorList
Enumeration names = eList.getAllNames();

//create two new vectors
Vector data = new Vector();
Vector errors = new Vector();

//loop over the number of Errors
while (names.hasMoreElements()) {
    //add all the errors to the vector error
    //and the names assoc. with the errors to the vector data
    String s = (String) names.nextElement();
    Vector v = (Vector) eList.get(s);
    errors.addElement(v);
    data.addElement(new ErrorData(s));
}

//clear the current screen
content.removeAll();

//create a new layout
BorderLayout border = new BorderLayout();
content.setLayout(border);

// Set the horizontal gap between components
border.setHgap(20);

//create new border of type BevelBorder
BevelBorder edge = new BevelBorder(BevelBorder.RAISED);

//establish 4 new panels
JPanel topPanel, mainPanel, leftPanel, rightPanel;

//add the topPanel to the content
content.add(topPanel = new JPanel(), BorderLayout.NORTH);

//makes a rectangular grid of 2 rows and 1 column
topPanel.setLayout(new GridLayout(2,1));

//creates two new panels
JPanel titlePanel, infoPanel;

//add the titlePanel to the top Panel with a center layout
topPanel.add(titlePanel = new JPanel());
titlePanel.setLayout(new FlowLayout(FlowLayout.CENTER));

//add the infoPanel to the top Panel with a center layout
topPanel.add(infoPanel = new JPanel());
infoPanel.setLayout(new FlowLayout(FlowLayout.CENTER));

//add to titlePanel the title Project Errors
JLabel titleLabel;
titlePanel.add(titleLabel = new JLabel(createTitle("Project Errors")));
titleLabel.setFont(new Font("SanSerif", Font.BOLD, 18));

//add the label to the info Panel
infoPanel.add(new JLabel("Select the function from the list on the left to view its errors"));

m_dataError = new ErrorTableData(data, errors);

//new table instance
table = new JTable();

//set's table selection at a single selection
ListSelectionModel.SINGLE_SELECTION);
//set size of viewport of table
Dimension(content.getWidth()/4, content.getHeight()-150));
//establisshes that the table cannot be resized
AUTO_RESIZE_OFF);

//cannot create columns by self
setAutoCreateColumnsFromModel(false);
//sets the model for the table to be m_dataError (which contains all the data and assocaited errors)
setModel(m_dataError);

//loop over the total number of cols.
for (int k=0; k < ErrorTableData.m_columns.length; k++) {
    //access an invididual cell of the table
    DefaultTableCellRenderer renderer = new
    DefaultTableCellRenderer();
    //sets the alignment of the columns to be horizontal
    renderer.setHorizontalAlignment(ErrorTableData.m_columns[k].m_alignment );
    //create a new column with index at k, width of column
    //basically, a column is created with index at k.
    TableColumn column = new TableColumn(k,
    ErrorTableData.m_columns[k].m_width, renderer, null);
    column.setMinWidth(ErrorTableData.m_columns[k].m_width);
    //add the column to the table
    table.addColumn(column);
}

//sets properties of header
JTableHeader header = table.getTableHeader();
header.setUpdateTableInRealTime(false);
header.setReorderingAllowed(false);
//table.getColumnModel().addColumnModelListener (mdataError.new ColumnMovementListener());

//add mainPanel with a center layout to content
content.add(mainPanel = new JPanel(), BorderLayout.CENTER);
mainPanel.setLayout(new BorderLayout());

//add left/right panel to mainPanel
mainPanel.add(leftPanel = new JPanel(), BorderLayout.WEST);
mainPanel.add(rightPanel = new JPanel(), BorderLayout.CENTER);

//sets the Border for the top/mainPanel to be BevelBorder (which is edge)
topPanel.setBorder(edge);
mainPanel.setBorder(edge);

//Create the scroll pane and add the table to it.
JScrollPane scrollPane = new JScrollPane(table);

//Add the scroll pane to this window.
leftPanel.add(scrollPane, BorderLayout.CENTER);
scrollPane.setVisible(true);

//creates area of Error Window to have 33 rows and 62 columns
JTextArea errorWin = new JTextArea(33,62);
//add a scrollable view to the errorWin
JScrollPane errorPane = new JScrollPane(errorWin);
//the ErrorWindow cannot be edited by the user
errorWin.setEditable(false);

//set the border of the ErrorPane to be BevelBorder (which is edge)
errorPane.setBorder(edge);

//set the size of the errorWin
errorWin.setSize((new Double(75*content.getWidth()).intValue(), content.getHeight()-150);
//set the border of the errorWin to be BevelBorder (which is edge)
errorWin.setBorder(edge);

//add the errorPane to the rightPanel
rightPanel.add(errorPane);

//add a MouseListenter to the table - meaning that the table will receive "mouse" events from m_dataError (which contains all the errors)
//In addition, m_dataError will receive events from the "column" events from the table and error window
//in essence each column of the abl and error window can be accessed by clicking the mouse
table.addMouseListener(m_dataError.new ColumnListener(table, errorWin));

repaint();
} //closes method showErrorTable
/** @requires Vector errors !=null  
 * @modifies (JTable table) by adding viewing properties to the table  
 * @modifies (Container content) by adding a JScrollPane  
 * @effects creates the Error table which contains a list of the function names  
 */  
private void showErrors(Vector errors) {  
  //modes contains all of the elements of the errors vector  
  Enumeration mods = errors.elements();  
  //new vector called names is created  
  Vector names = new Vector();  
  
  //loop over number of errors  
  while (mods.hasMoreElements()) {  
    //add the error to the names vector  
    names.addElement((String) mods.nextElement());  
  } //closes while-loop  
  content.removeAll(); //clear the current screen  
  //makes a new table with the title errors and displays the names of all the functions  
  MyTableModel myModel = new MyTableModel("Errors", names);  
  //new instance of a table  
  table = new JTable(myModel);  
  //set's table selection at a single selection  
  table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);  
  //set size of viewport of table  
  table.setPreferredScrollableViewportSize(new Dimension(500, 70));  
  //Create the scroll pane and add the table to it.  
  JScrollPane scrollPane = new JScrollPane(table);  
  //Add the scroll pane to this window.  
  content.add(scrollPane);  
  scrollPane.setVisible(true);  
  scrollPane.setBounds(0,0,content.getWidth()/2,content.getHeight());  
  repaint();  
} //closes method showErrors  
/** @requires String s !=null  
 * @effects tokenizes a string and returns it as a string buffer with the spaces separated by " "  
 */  
public static String formatForTable(String s) {  
  //st contains the tokens of String s  
  StringTokenizer st = new StringTokenizer(s);  
  StringBuffer sb = new StringBuffer();  
  
  //loop over the tokens  
  while (st.hasMoreTokens()) {  
    sb.append(st.nextToken() + " "); //append each each token to  
  } //close while-loop

223
/**
   @requires String q1, q2 !=null
   @effects allows the user to select out two functions and returns
   them in array modName
*/
private String[] selectTwoFunctions(String q1, String q2) {

    //mods contains all of the Function Names
    Enumeration mods = client.getFunctionsNames();
    //new vector names is created
    Vector names = new Vector();
    //modName array of two elements is created
    String[] modName = new String[2];

    //loop over the number of functions
    while (mods.hasMoreElements()) {
        //add the name of each function to the names vector
        names.addElement((String) mods.nextElement());
    } //closes while-loop

    //create new array choices which holds the number of functions
    //as determined by names.size()
    Object[] choices = new Object[names.size()];

    //loop over the number of functions
    for(int i=0; i < names.size(); i++) {
        //put each of the functions in the choices array
        choices[i] = names.elementAt(i);
    } //closes for-loop

    //first element of ModName is a function that the user selects
    from choices
    modName[0] = (String) JOptionPane.showInputDialog(mf, q1, "Select
Function", JOptionPane.QUESTION_MESSAGE, null, choices, choices[0]);

    //put all the names of the functions in mods
    mods = client.getFunctionsNames();
    //create new vector names
    names = new Vector();

    //loop over the number of functions
    while (mods.hasMoreElements()) {
        //add all functions to names which are not in modName already
        String m = (String) mods.nextElement();
        if (!m.equals(modName[0]))
            names.addElement(m);
    } //close while-loop

    //create new array choices which holds the number of functions
    //as determined by names.size()
    choices = new Object[names.size()];

    //loop over the number of functions
    for(int i=0; i < names.size(); i++) {

//out each function into the choices array
choices[i] = names.elementAt(i);
}
//have the user select another function, which is the second
element of modName
modName[i] = (String) JOptionPane.showInputDialog(mf, q2, "Select
Function", JOptionPane.QUESTION_MESSAGE, null, choices, choices[0]);

return modName; //return modName, which contains two functions
} //closes method selectTwoFunctions

/** @effects a function is selected depending on if the user is in
the table with a function or not */
private String selectFunction() {
  //modName is null
  String modName = null;
  //if the table is null or user is in column Function Name or if
the user is not in a selected row, then list all the functions
  if ((table == null) || (!table.getColumnName(0).equals("Function
Name")) || (table.getSelectedRow() == -1)) { //
    //showFunctions();
    //mods lists all the names of the Functions
    Enumeration mods = client.getFunctionsNames();
    //create a new vector names
    Vector names = new Vector();
    //loop over the functions
    while (mods.hasMoreElements()) {
      //add the name of each function to the Names vector
      names.addElement((String) mods.nextElement());
    } //close the while-loop
    //array choices has size determined by the number of
    functions
    Object[] choices = new Object[names.size()];
    //loop over the number of functions
    for(int i=0; i < names.size(); i++) {
      //choices contains the names of all the functions
      choices[i] = names.elementAt(i);
    } //closes for-loop
    //modName contains the one function that is selected
    modName = (String) JOptionPane.showInputDialog(mf, "Select a
Function: ", "Select Function", JOptionPane.QUESTION_MESSAGE, null, choices, choices[0]);
    } //closes the if-statement

  else { //if user is already in a function column
    //get the row the user is in
    int row = table.getSelectedRow();
    //get the function name at that row and column 0
    modName = (String) table.getValueAt(row, 0);
  } //closes else

return modName; //return the function
private String selectDesignVar() {
  String dv = null;
  if ((table == null) || (!table.getColumnModel().getColumn(0).equals("Design Variable Name")) || (table.getSelectedRow() == -1)) {
    Collection dvs = client.getDesignVars();
    String[] choices = new String[dvs.size()];

    Iterator i = dvs.iterator();
    int count = 0;

    while (i.hasNext()) {
      DesignVar d = (DesignVar) i.next();
      choices[count] = d.getName();
      count++;
    }
    dv = (String)JOptionPane.showInputDialog(null, "Select a Design Variable:", "Design Variable", JOptionPane.QUESTION_MESSAGE, null, choices, choices[0]);
  } else {
    dv = (String) table.getValueAt(table.getSelectedRow(),0);
  }
  return dv;
}

/**
   * @param dialogTitle, approveButtonText, approveButtonTooltip !=null
   * @param char approveButtonMnemonic, File file !=null
   * @param (JFileChooser files) by setting the dialog test,
   * approveButtonText, ApproveButtonToolTipText,
   * approvedButtonMnemonic, SelectedFile, rescanning the current
directory, showing Dialog
   * @effects allows the user to select a file and shows
   */
private File showDialog(String dialogTitle, String approveButtonText, String approveButtonTooltip, char approveButtonMnemonic, File file) {
  files.setDialogTitle(dialogTitle); //set the dialog title
  files.setApproveButtonText(approveButtonText); //sets text used
  files.setApproveButtonToolTipText(approveButtonTooltip); //sets text used
  files.setFileSelectionMode(files.FILES_ONLY); //allows user to
  files.rescanCurrentDirectory(); //rescan current directory
  files.setSelectedFile(file); //sets the selected file to be the
  return files.getFileSelectionMode();
}
//pops up a custom file chooser dialog, which result is bound to int result = files.showDialog(mf, null);
//if result is approved, then get the selected file, else do nothing
return (result == files.APPROVE_OPTION) ? files.getSelectedFile() : null;
/**
@requires String heading, info, title, int numChars !=null
* @effects shows the Long Dialog of the String info
*
private void showLongDialog(String heading, String info, String title, int numChars) {
    char[] cArr = info.toCharArray(); //converts info String into a character Array
    char[] newArr = new char[cArr.length]; //newArr holds the number of elements in cArr
    boolean insertBreak = false; //sets insertbreak to be false

    //loop over the Character Array
    //entire loop (with if's) will add every character from cArr into newArr
    for(int i=0; i < cArr.length; i++) {
        if (((i % numChars) == 0) && (i != 0)) {
            insertBreak = true;
        }
        if ((insertBreak) && (cArr[i] == ' ')) {
            insertBreak = false;
            newArr[i] = '\n';
        } else {
            newArr[i] = cArr[i];
        } //closes else
    } //closes for-loop
    String newInfo = new String(newArr); //newInfo has the String elements of newArr
    //brings up a dialog that displays the newInfo
    JOptionPane.showMessageDialog(mf, heading + newInfo, title,
    JOptionPane.INFORMATION_MESSAGE);
} //close showLongDialog
/**
@modifies (JMenu ProjectMenu, moduleMenu, checkMenu) to be enabled
* @modifies (JMenuItem saveProjectItem, closeProjectAsItem) to be enabled
* @effects opens a project and sets many menu items to be enabled
*/
private void projectOpen() {
    projectMenu.setEnabled(true);
    moduleMenu.setEnabled(true);
    checkMenu.setEnabled(true);
    saveProjectItem.setEnabled(true);
    saveProjectAsItem.setEnabled(true);
    closeProjectItem.setEnabled(true);
} //closes method projectOpen
/** @modifies (JMenu ProjectMenu, moduleMenu, checkMenu) to be
disenabled
* @modifies (JMenuItem saveProjectItem, closeProjectAsItem) to be
disenabled
* @effects closes a project and sets many menu items to be
disenabled
*/
private void projectClosed()
{
projectMenu.setEnabled(false);
moduleMenu.setEnabled(false);
checkMenu.setEnabled(false);
openProjectItem.setEnabled(true);
saveProjectItem.setEnabled(false);
saveProjectAsItem.setEnabled(false);
closeProjectItem.setEnabled(false);
} //closes method projectClose

/** @requires Variable v !=null
* @effects displays to the user that the data for a specific
variable is being edited
*/
public void updateInfo(Variable v) {
System.out.println("editing data for " + v.getName());
} //closes method UpdateInfo

/** @requires Vector data, variables, String title !=null
* @modifies (VariableTableData m_data)by adding data and
variables
* @modifies (JTable table) by adding viewing preferences and
columns
* @effects displays the topPanel and buttons for viewing/not-
viewing
* certain columns of the table
*/
private JPanel createTopPanel(Vector data, Vector variables, String title) {

//create a new instance of a JPanel called topPanel
JPanel topPanel = new JPanel();

//set the layout to be a rectangular grid of 3 rows and 1 column
topPanel.setLayout(new GridLayout(3, 1));

//create three other JPanel variables
JPanel buttonPanel, checkBoxPanel, titlePanel;

//set the edge to be BevelBorder
BevelBorder edge = new BevelBorder(BevelBorder.RAISED);

//add the titlePanel to the topPanel and create a central layout
topPanel.add(titlePanel = new JPanel());
titlePanel.setLayout(new FlowLayout(FlowLayout.CENTER));

//add the ButtonPanel to the topPanel and create a central layout
topPanel.add(buttonPanel = new JPanel());
buttonPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
//add the checkBoxPanel to the topPanel and create a central layout
topPanel.add(checkBoxPanel = new JPanel());
checkBoxPanel.setLayout(new FlowLayout(FlowLayout.CENTER));

//create a title Label with SanSerif font
JLabel titleLabel;
titlePanel.add(titleLabel = new JLabel(createTitle(title)));
titleLabel.setFont(new Font("SanSerif", Font.BOLD, 18));

//br is a new variable of type ButtonReaction
ButtonReaction br = new ButtonReaction();

// CREATE THE BUTTONS
//create View Rationale button, sets its border and allows it to receive action events
buttonPanel.add(ratButton = new JButton("View Rationale"));
ratButton.setBorder(edge);
ratButton.addActionListener(br);

//create Editbutton, sets its border and allows it to receive action events
buttonPanel.add(editButton = new JButton("Edit Info"));
editButton.setBorder(edge);
editButton.addActionListener(br);

//m_data holds the data and variables
m_data = new VariableTableData(data, variables);

//create a new instance of a table
table = new JTable();

//set's table selection at a single selection
table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

//set size of viewport of table
table.setPreferredScrollableViewportSize(new Dimension(content.getWidth(-50, content.getHeight(-150))));

//establishes that the table cannot be resized
table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);

//cannot create columns by self
table.setAutoCreateColumnsFromModel(false);

//sets the model for the table to be m_data (which contains all the data and variables)
table.setModel(m_data);

//loop over the number of columns in the VariableTableData
for (int k=0; k < VariableTableData.m_columns.length; k++) {
    //access an invididual cell of the table
    DefaultTableCellRenderer renderer = new DefaultTableCellRenderer();
    //sets the alignment of the columns to be horizontal
    renderer.setHorizontalAlignment(VariableTableData.m_columns[k].m_alignment);
//create a new column with index at k, width of column depending upon element k, the cell is rendered and cannot be edited
//basically, a column is created with index at k.
TableColumn column = new TableColumn(k,
VariableTableData.m_columns[k].m_width, renderer, null);
column.setMinWidth(VariableTableData.m_columns[k].m_width);
//add the column to the table
table.addColumn(column);
}

//creates a header and establishes its properties
JTableHeader header = table.getTableHeader();
header.setUpdateTableInRealTime(false);
header.setReorderingAllowed(false);

//the columns of the table are now listeners
table.getColumnModel().addColumnModelListener (m_data.new ColumnMovementListener());

//model is bound to all the data contained in the columns
TableColumnModel model = table.getColumnModel();

//create the checkboxes which hold 7 elements
JCheckBox[] checkBox = new JCheckBox[7];

//add Name checkbox to checkBoxPanel and allows it to receive action events
//such that if it is clicked, the name column will be displayed in the table
checkBoxPanel.add(checkBox[0] = new JCheckBox("Name", true));
checkBox[0].addActionListener(new ColumnKeeper(model.getColumn(0), VariableTableData.m_columns[0]));

//add Value checkbox to checkBoxPanel and allows it to receive action events
//such that if it is clicked, the value column will be displayed in the table
checkBoxPanel.add(checkBox[1] = new JCheckBox("Value", true));
checkBox[1].addActionListener(new ColumnKeeper(model.getColumn(1), VariableTableData.m_columns[1]));

//add Units checkbox to checkBoxPanel and allows it to receive action events
//such that if it is clicked, the units column will be displayed in the table
checkBoxPanel.add(checkBox[2] = new JCheckBox("Units", false));
checkBox[2].addActionListener(new ColumnKeeper(model.getColumn(2), VariableTableData.m_columns[2]));

//add Range checkbox to checkBoxPanel and allows it to receive action events
//such that if it is clicked, the range column will be displayed in the table
checkBoxPanel.add(checkBox[3] = new JCheckBox("Range", false));
checkBox[3].addActionListener(new ColumnKeeper(model.getColumn(3), VariableTableData.m_columns[3]));
//add Author Checkbox to checkBoxPanel and allows it to receive action events
//such that if it is clicked, the author column will be displayed in the table
checkBoxPanel.add(checkBox[4] = new JCheckBox("Author", false));
checkBox[4].addActionListener(new ColumnKeeper(model.getColumn(4), VariableTableData.m_columns[4]));

//add Date Checkbox to checkBoxPanel and allows it to receive action events
//such that if it is clicked, the date column will be displayed in the table
checkBoxPanel.add(checkBox[5] = new JCheckBox("Date", false));
checkBox[5].addActionListener(new ColumnKeeper(model.getColumn(5), VariableTableData.m_columns[5]));

//add Aliases Checkbox to checkBoxPanel and allows it to receive action events
//such that if it is clicked, the aliases column will be displayed in the table
checkBoxPanel.add(checkBox[6] = new JCheckBox("Aliases", false));
checkBox[6].addActionListener(new ColumnKeeper(model.getColumn(6), VariableTableData.m_columns[6]));

//removes all columns from the table except for the name and value column
//(which are originally displayed when the variable data is viewed)
for (int k=2; k < VariableTableData.m_columns.length; k++) {
table.removeColumn(table.getColumnModel().getColumn(2));
} //closes for-loop
return topPanel;

/** @effect method used to create test data.
  * it creates new constants, adds them to constantlists c1, c2, c3
  * then adds these lists to Vector v
  */
private Vector createConstantListVector() {
  Vector v = new Vector(); //creates a new Vector v
  ConstantList c1 = new ConstantList(); //creates three new
  ConstantList c2 = new ConstantList();
  ConstantList c3 = new ConstantList();

  c1.add("speed", new Constant("speed", "80", ",", ",", ",", ","));
  c2.add("power", new Constant("power", "5000", ",", ",", ",", ","));
  c3.add("torque", new Constant("torque", "350", ",", ",", ",", ","));
}
c2.add("speed", new Constant("speed", "80", "", "", "", "");
//adds constant speed to c2

c2.add("power", new Constant("power", "4000", "", "", "", "");
//adds constant power to c2

c2.add("torque", new Constant("torque", "375", "", "", "", "");
//adds constant torque to c2

c2.add("legs", new Constant("legs", "3", "", "", "", "");
//adds the constant legs to c2

c3.add("speed", new Constant("speed", "85", "", "", "", "");
//adds constant speed to c3

c3.add("power", new Constant("power", "4000", "", "", "", "");
//adds constant power to c3

c3.add("torque", new Constant("torque", "375", "", "", "", "");
//adds constant torque to c3

c3.add("legs", new Constant("legs", "3", "", "", "", "");

//adds constant lists to Vector v
v.addElement(c1);
v.addElement(c2);
v.addElement(c3);

return v; //returns the vector

} //closes method createConstants

//-----------------------
// PRIVATE CLASSES
//-----------------------

class MyTableModel extends AbstractTableModel {

final String[] columnNames;
final Object[][] data;

MyTableModel(String colName, Vector columnData) {
    columnNames = new String[1];
    columnNames[0] = colName;
    data = new Object[columnData.size()][1];
    for(int i=0; i < columnData.size(); i++) {
        data[i][0] = columnData.elementAt(i);
    }
}

MyTableModel(String col1, Vector data1, String col2, Vector data2) {
    columnNames = new String[2];
    columnNames[0] = col1;
    columnNames[1] = col2;
    data = new Object[data1.size()][2];
    for(int i=0; i < data1.size(); i++) {
        data[i][0] = data1.elementAt(i);
    }
}
public int getColumnCount() {
    return columnNames.length;
}

public int getRowCount() {
    return data.length;
}

public String getColumnName(int col) {
    return columnNames[col];
}

public Object getValueAt(int row, int col) {
    return data[row][col];
}

/*
 * JTable uses this method to determine the default renderer/
 * editor for each cell. If we didn't implement this method,
 * then the last column would contain text ("true"/"false"),
 * rather than a check box.
 */
public Class getColumnClass(int c) {
    return getValueAt(0, c).getClass();
}

/*
 * Don't need to implement this method unless your table's
 * editable.
 */
public boolean isCellEditable(int row, int col) {
    //Note that the data/cell address is constant,
    //no matter where the cell appears onscreen.
    if (col < 2) {
        return false;
    } else {
        return true;
    }
}

/*
 * Don't need to implement this method unless your table's
 * data can change.
 */
public void setValueAt(Object value, int row, int col) {
    data[row][col] = value;
    fireTableCellUpdated(row, col);
}
} //closes class MyTableModel
public class MyFileFilter extends javax.swing.filechooser.FileFilter {

    String extension, description;

    MyFileFilter(String e, String desc) {
        extension = e;
        description = desc;
    }

    // Accept all directories and all gif, jpg, or tiff files.
    public boolean accept(File f) {
        if (f.isDirectory()) {
            return true;
        }

        String ext = null;
        String s = f.getName();
        int i = s.lastIndexOf('.' +

        if (i > 0 && i < s.length() - 1) {
            ext = s.substring(i+1).toLowerCase();
        } else {
            return false;
        }

        return false;
    }

    // The description of this filter
    public String getDescription() {
        return description;
    }
} // closes class MyFileFilter

class ColumnKeeper implements ActionListener {

    protected TableColumn m_column;
    protected ColumnData m_colData;

    public ColumnKeeper(TableColumn column, ColumnData colData) {
        m_column = column;
        m_colData = colData;
    }

    public void actionPerformed(ActionEvent e) {
        JCheckBox item = (JCheckBox) e.getSource();
        TableColumnModel model = table.getColumnModel();
    }
}
if (item.isSelected()) {
    model.addColumn(m_column);
} else {
    model.removeColumn(m_column);
}

table.tableChanged(new TableModelEvent(m_data));
table.repaint();

I
//closes class ColumnKeeper

class ButtonReaction implements ActionListener {
    public ButtonReaction() {};
    public void actionPerformed(ActionEvent e) {
        JButton item = (JButton) e.getSource();
        if (item == ratButton) {
            if (table.getSelectedRow() == -1) {
                JOptionPane.showMessageDialog(mf, "You must select a row in the table in order to view the rationale", "Error", JOptionPane.ERROR_MESSAGE);
            } else {
                JOptionPane.showMessageDialog(mf, m_data.getVariableAt(table.getSelectedRow().getRationale(), "Rationale", JOptionPane.INFORMATION_MESSAGE);
            }
        } else {
        }
    }
    if (item == editButton) {
        if (table.getSelectedRow() == -1) {
            JOptionPane.showMessageDialog(mf, "You must select a row in the table in order to edit the info", "Edit Error", JOptionPane.ERROR_MESSAGE);
        } else {
            Variable var = m_data.getVariableAt(table.getSelectedRow());
            editPane = new RationaleDialogGUI(mf,"fake", var);
            //editPane = new RationaleDialogGUI("fake");
            editPane.setVisible(true);
        }
    }
}
class ColumnData {
    public String m_title;
    public int m_width;
    public int m_alignment;

    public ColumnData(String title, int width, int alignment) {
        m_title = title;
        m_width = width;
        m_alignment = alignment;
    }
}

class VariableTableData extends AbstractTableModel {
    static final public ColumnData m_columns[] = {
        new ColumnData("Name", 150, JLabel.LEFT),
        new ColumnData("Value", 200, JLabel.LEFT),
        new ColumnData("Units", 100, JLabel.LEFT),
        new ColumnData("Valid Range", 150, JLabel.LEFT),
        new ColumnData("Author", 100, JLabel.LEFT),
        new ColumnData("Date", 100, JLabel.LEFT),
        new ColumnData("Aliases", 150, JLabel.LEFT),
    };

    protected Vector m_vector;
    protected Vector m_vars;
    protected int m_columnsCount = m_columns.length;

    public VariableTableData(Vector data, Vector vars) {
        m_vector = data;
        m_vars = vars;
    }

    public int getRowCount() {
        return m_vector == null ? 0 : m_vector.size();
    }

    public int getColumnCount() {
        return m_columnsCount;
    }

    public String getColumnName(int column) {
        return m_columns[column].m_title;
    }

    public boolean isCellEditable(int nRow, int nCol) {
        return false;
    }
}
public Variable getVariableAt(int nRow) {
    return (Variable) m_vars.elementAt(nRow);
}

public Object getValueAt(int nRow, int nCol) {
    if (nRow < 0 || nRow >= getRowCount())
        return "";

    VariableData row = (VariableData) m_vector.elementAt(nRow);
    switch (nCol) {
        case 0: return row.m_name;
        case 1: return row.m_value;
        case 2: return row.m_units;
        case 3: return row.m_range;
        case 4: return row.m_author;
        case 5: return row.m_date;
        case 6: return row.m_aliases;
        /*String s = "";
         Vector v = row.m_aliases;

         for(int i=0; i < v.size(); i++) {
             s += v.elementAt(i).toString();
             if (i != v.size()-1) {
                 s += ", ";
             }
         }
         return s;*/
        return "";
    }

    return "";
}

public String getTitle() {
    return "My title";
}

//-------------------------------
// INTERNAL CLASSES TO
// CLASS VARIABLETABLEDATA
//-------------------------------

class ColumnListener extends MouseAdapter {
    protected JTable table;

    public ColumnListener(JTable t) {
        table = t;
    }

    public void mouseClicked(MouseEvent e) {
        TableColumnModel colModel = table.getColumnModel();
        int columnModelIndex = colModel.getColumnIndexAtX(e.getX());
    }
}
int modelIndex = 
colModel.getColumn(columnModelIndex).getModelIndex();

for(int i=0; i < m_columnsCount; i++) {
    TableColumn column = colModel.getColumn(i);
    column.setHeaderValue(getColumnName(column.getModelIndex()));
}

class ColumnMovementListener implements TableColumnModelListener {
    public void columnAdded(TableColumnModelEvent e) {
        m_columnsCount++;
    }
    public void columnRemoved(TableColumnModelEvent e) {
        m_columnsCount--;
    }
    public void columnMarginChanged(ChangeEvent e) {} 
    public void columnMoved(TableColumnModelEvent e) {} 
    public void columnSelectionChanged(ListSelectionEvent e) {} 
}

class VariableTableData {
    public String m_name;
    public String m_value;
    public String m_units;
    public String m_range;
    public String m_author;
    public Date m_date;
    public String m_aliases;

    public VariableTableData(String name, String value, String units, String range, String author, Date date, String aliases) {
        m_name = name;
        m_value = value;
        m_units = units;
        m_range = range;
        m_author = author;
        m_date = date;
        m_aliases = aliases;
    }
}

class VariableData {
    public String m_name;
    public String m_value;
    public String m_units;
    public String m_range;
    public String m_author;
    public Date m_date;
    public String m_aliases;
class HistoryTableData extends AbstractTableModel {

    protected Vector m_vector;
    protected int m_columnsCount;

    public HistoryTableData(Vector data, Vector v) {
        m_vector = data;
        m_columnsCount = v.size();
    }

    public int getRowCount() {
        return m_vector == null ? 0 : m_vector.size();
    }

    public int getColumnCount() {
        return m_columnsCount;
    }

    public String getColumnName(int column) {
        if (column == 0) {
            return "Name";
        } else {
            return "Value (t = " + column + ")";
        }
    }

    public boolean isCellEditable(int nRow, int nCol) {
        return false;
    }

    public Object getValueAt(int nRow, int nCol) {
        if (nRow < 0 || nRow >= getRowCount())
            return "";

        HistoryData row = (HistoryData) m_vector.elementAt(nRow);
        if (nCol == 0) {
            return row.m_name;
        } else {
            return row.m_value[nCol-1];
        }
    }

    public String getTitle() {
        return "My title";
    }
}

// closes HistoryTableData

class HistoryData {

    public String m_name;
    public ColorData[] m_value;
}
public HistoryData(Variable var, Vector vec) {

    m_value = new ColorData[vec.size()];
    m_name = var.getName();

    TimeList prevList = new TimeList();
    TimeList currList = new TimeList();

    for (int i = 0; i < vec.size(); i++) {
        currList = (TimeList) vec.get(i);
        Variable curr = (Variable) currList.get(m_name);

        if (i == 0) {
            if (curr == null) {
                m_value[i] = new ColorData(Color.white, ");
            } else {
                m_value[i] = new ColorData(Color.white,
                MainFrame.formatForTable(curr.getValue());
            }
        } else {

            prevList = (TimeList) vec.get(i-1);
            if (currList.contains(m_name)) {
                Variable prev = (Variable) prevList.get(m_name);

                Color thisColor;

                // Variable added
                if (prev == null) {
                    thisColor = Color.green;
                } // Variable removed
                else if (curr == null) {
                    thisColor = (prev == null) ? Color.white : Color.lightGray;
                // Variable changed
                } else if (!curr.getValue().equals("")) &&
                (!curr.getValue().equals(prev.getValue())) {
                    thisColor = Color.red;
                } else {
                    thisColor = Color.white;
                }

                m_value[i] = new ColorData(thisColor,
                MainFrame.formatForTable(curr.getValue()));

                // Variable was removed at this stage
            } else {
                m_value[i] = (prevList.contains(m_name)) ? new
                ColorData(Color.lightGray, "); new ColorData(Color.white, "");
            }
        }
    }
}
class ColoredTableCellRenderer extends DefaultTableCellRenderer {
    public void setValue(Object value) {
        if (value instanceof ColorData) {
            ColorData cvalue = (ColorData) value;
            setBackground(cvalue.m_color);
            setText(cvalue.m_data.toString());
        } else {
            super.setValue(value);
        }
    }
} //closes ColoredTableCellRenderer

class ColorData {
    public Color m_color;
    public Object m_data;

    public ColorData(Color color, Object data) {
        m_color = color;
        m_data = data;
    }

    public String toString() {
        return m_data.toString();
    }
} //closes class ColorData

class ErrorTableData extends AbstractTableModel {
    static final public ColumnData m_columns[] = {
        new ColumnData("Function Name", 200, JLabel.LEFT),
    };

    protected Vector m_vector;
    protected Vector m_errors;
    protected int m_columnsCount = m_columns.length;

    public ErrorTableData(Vector data, Vector errors) {
        m_vector = data;
        m_errors = errors;
    }

    public int getRowCount() {
        return m_vector == null ? 0 : m_vector.size();
    }
}
public int getColumnCount() {
    return m_columnsCount;
}

public String getColumnName(int column) {
    return m_columns[column].m_title;
}

public boolean isCellEditable(int nRow, int nCol) {
    return false;
}

public Vector getErrorVectorAt(int nRow) {
    return (Vector) m_errors.elementAt(nRow);
}

public Object getValueAt(int nRow, int nCol) {
    if (nRow < 0 || nRow >= getRowCount())
        return "";
    ErrorData row = (ErrorData) m_vector.elementAt(nRow);
    switch (nCol) {
        case 0: return row.m_name;
    }
    return "";
}

public String getTitle() {
    return "My title";
}

//------------------------
// INTERNAL CLASSES TO
// CLASS ERRORTABLEDATA
//------------------------

class ColumnListener extends MouseAdapter {

    protected JTextArea text;
    protected JTable table;

    public ColumnListener(JTable tab, JTextArea t) {
        text = t;
        table = tab;
    }

    public void mouseClicked(MouseEvent e) {
        int row = table.getSelectedRow();
        Vector errorVec = getErrorVectorAt(row);
        String s = "";

        for(int i=0; i < errorVec.size(); i++) {
            Error err = (Error) errorVec.get(i);
            s += (i+1) + "  " + err.getMsg() + "\n\n";
        }
    }
}
class ErrorData {
    public String m_name;

    public ErrorData(String name) {
        m_name = name;
    }
}

class FuncIO extends Variable {
    public FuncIO(String s, String val) {
        super(s, val, "", ",", ",", "");
    }
}

//closes class ErrorTableData
//closes class ColumnListener
10.2.3 Utilities

Written by Shane Cruz [21].

//package SSPARC;

import java.awt.event.ActionEvent;
import java.awt.*;

public class Utilities {
    public static void openFile(Frame f) {
        FileDialog openFileDialog1 = new FileDialog(f);
        openFileDialog1.setMode(FileDialog.LOAD);
        openFileDialog1.setTitle("Open");

        try {
            int defMode = openFileDialog1.getMode();
            String defTitle = openFileDialog1.getTitle();
            String defDirectory = openFileDialog1.getDirectory();
            String defFile = openFileDialog1.getFile();

            openFileDialog1 = new FileDialog(f, defTitle, defMode);

            openFileDialog1.setDirectory(defDirectory);
            openFileDialog1.setFile(defFile);
            openFileDialog1.setVisible(true);
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
10.2.4 FileSaver

Written by Shane Cruz [21].

package SSPARC;
import java.io.*;

public class FileSaver {

    // This function saves an object to a file.
    public static void saveObjectInFile(Object toBeSaved, String filename)
        throws ClassNotFoundException, IOException {

        // creates an output stream to a file
        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(filename + ".ssp"));

        // writes this object to the file
        out.writeObject(toBeSaved);

        // closes the output and flushed it.
        out.flush();
        out.close();
    }

    // this function takes in a filename and loads the object that was saved in
    // that file.
    public static Object loadFileObject(String filename)
        throws ClassNotFoundException, IOException {

        // creates an input stream to the given file
        ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename + ".ssp"));

        // returns the object in the file
        return in.readObject();
    }
}
10.2.5 RationaleDialogGUI

Written by Presley Cannady [22].

/**
 * @author "P.H. Cannady" <revprez@mit.edu>
 */
package SSPARC;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.beans.*;

/**
 * OVERVIEW - RationaleDialogGUI extends JDialog and is a custom
dialog that provides textfields for entering values
mapped to name, author, valid range, and rationale
Additionally, rationale textfields can be expanded
to textareas if necessary
*
* @specfield NAME : String // caption text for name field
* @specfield VALUE : String // caption text for value field
* @specfield UNITS : String // caption text for units field
* @specfield VALIDRANGE : String // caption text for valid range field
* @specfield AUTHOR : String // caption text for author field
* @specfield ALIASES : String // caption text for aliases field
* @specfield RATIONALE : String // caption text for rationale field
* @specfield MORE : String // text for more button
* @specfield namef : JTextField // name field
* @specfield valuef : JTextField // value field
* @specfield unitsF : JTextField // units field
* @specfield valid-rangef : JTextField // valid range field
* @specfield authorf : JTextField // author field
* @specfield aliasesf : JTextField // aliases field
* @specfield rationale_f_a : JTextArea // rationale text area
* @specfield name : String // the name
* @specfield value : String // the value
* @specfield units : String // the units
* @specfield valid-range : String // the valid range
* @specfield author : String // the author
* @specfield aliases : String // the aliases
* @specfield rationale : String // the rationale
* @specfield more : JButton // the more button (open text area for rationale)
* @specfield moreArea : boolean // state of rationale field/textarea
* @specfield optionPane : JOptionPane // the option pane containing the fields
* @endspec
*/

public class RationaleDialogGUI extends JDialog {

    public static final String NAME = "name";

    public static final String NAME = "name";
public static final String VALUE = "value";
public static final String UNITS = "units";
public static final String VALID_RANGE = "valid range";
public static final String AUTHOR = "author";
public static final String ALIASES = "aliases";
public static final String RATIONALE = "rationale";
public static final String MORE = "Rationale";

private final JTextField name_f = new JTextField(20);
private final JTextField value_f = new JTextField(20);
private JTextField units_f;
private JTextField validrange_f;
private JTextField author_f;
private JTextField aliases_f;
private JTextArea rationale_f_a;

private String name;
private String value;
private String units;
private String validrange;
private String author;
private String aliases;
private String rationale;
private final JButton more = new JButton("Rationale");

private boolean moreArea;
private JOptionPane optionPane;
private Variable var;

final MainFrame mf2;

/** @requires String title != null 
 * @effects instanciates a new RationaleDialogGUI */

public RationaleDialogGUI(Frame mf, String title, Variable var) {

super(mf);
 mf2 = (MainFrame) mf;
setTitle(title);
this.var = var;
name_f.setEditable(false);
value_f.setEditable(false);

units_f = new JTextField(var.getUnits(),20);
valid_range_f = new JTextField(var.getValidRange(),20);
author_f = new JTextField(var.getAuthor(),20);
aliases_f = new JTextField(var.getAliases(),20);
rationale_f_a = new JTextArea(var.getRationale(),20,40);

this.moreArea = false;
Object[] array = {NAME,
                name_f,
                VALUE,
                value_f,
                UNITS,
                units_f,
final String btnString1 = "Enter";
final String btnString2 = "Cancel";

Object[] options = {btnString1, btnString2};

OptionPane = new JOptionPane(array,
   JOptionPane.QUESTION_MESSAGE,
   JOptionPane.YES_NO_OPTION,
   null,
   options,
   options[0]);

setContentPane(optionPane);
setDefaultCloseOperation(DO NOTHING ON CLOSE);

final RationaleDialogGUI parent = this;

addWindowListener(new WindowAdapter() {
   **@requires WindowEvent we != null
   @effects closes out the System when window is killed
   */
   public void windowClosing(WindowEvent we) {
      optionPane.setValue(new Integer(JOptionPane.CLOSED_OPTION));
   }
});

name_f.addActionListener(new ActionListener() {
   **@requires ActionEvent e != null
   @modifies this (this.optionPane)
   @effects maps the String NAME to the option
   */
   private String name = null;

   public void actionPerformed(ActionEvent e) {
      optionPane.setValue(btnString1);
      name = name_f.getTexto;
      System.out.println(name);
   }

   public String getField() {
      return this.name;
   }
});

value_f.addActionListener(new ActionListener() {
   **@requires ActionEvent e != null
   @modifies this (this.optionPane)
   */
   public void actionPerformed(ActionEvent e) {
      value_f.setValueto(null);
      value_f = value_f_f;
   }
});
private String value = null;

public void actionPerformed(ActionEvent e) {
    optionPane.setValue(btnString1);
    value = value_f.getText();
    System.out.println(value);
}

public String getField() {
    return this.value;
}

units_f.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        optionPane.setValue(btnString1);
        units = units_f.getText();
        System.out.println(units);
    }
    public String getField() {
        return this.units;
    }
});

valid_range_f.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        optionPane.setValue(btnString1);
        valid_range = valid_range_f.getText();
        System.out.println(valid_range);
    }
    public String getField() {
        return this.valid_range;
    }
});

aliases_f.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        optionPane.setValue(btnString1);
        aliases = aliases_f.getText();
        System.out.println(aliases);
    }
    public String getField() {
        return this.aliases;
    }
});
private String author = null;

public void actionPerformed(ActionEvent e) {
    optionPane.setValue(btnString1);
    author = author_f.getText();
    System.out.println(author);
}

public String getField() {
    return this.author;
}
}

author_f.addActionListener(new ActionListener() {
    @requires ActionEvent e != null
    @modifies this (this.optionPane)
    @effects maps the String AUTHOR to the option
*/
private String aliases = null;
public void actionPerformed(ActionEvent e) {
    optionPane.setValue(btnString1);
    aliases = aliases_f.getText();
    System.out.println(aliases);
}

public String getField() {
    return this.aliases;
}
});

rationale_f_a.addPropertyChangeListener(new PropertyChangeListener() {
    @requires PropertyChangeEvent e != null
    @modifies this (this.optionPane)
    @effects grabs text from JTextArea rationale_f_a
*/
public void propertyChange(PropertyChangeEvent e) {
    rationale = rationale_f_a.getText();
}
});

more.addActionListener(new ActionListener() {

    @requires ActionEvent e != null
    @modifies this (this.optionPane)
    @effects pops up a text area dialog
*/
public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(parent, rationale_f_a, "Rationale", JOptionPane.QUESTION_MESSAGE);
}
optionPane.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent e) {
        String prop = e.getPropertyName();
        if (isVisible() && (e.getSource() == optionPane) &&
          prop.equals(JOptionPane.VALUEPROPERTY) ||
          prop.equals(JOptionPane.INPUT_VALUE_PROPERTY)) {
            Object val = optionPane.getValue();
            if (val == JOptionPane.UNINITIALIZED_VALUE) {
                //ignore reset
                return;
            }

            // Reset the JOptionPane's value.
            // If you don't do this, then if the user
            // presses the same button next time, no
            // property change event will be fired.
            optionPane.setValue(JOptionPane.UNINITIALIZED_VALUE);

            if (val.equals(btnStringValue)) {
                name = namef.getText();
                value = valuef.getText();
                units = unitsf.getText();
                valid_range = valid_rangep.getText();
                author = authorf.getText();
                aliases = aliasess.getText();

                if (rationale == null)
                    rationale = rationalef_a.getText();

                setVar(name,
                        value,
                        units,
                        valid_range,
                        author,
                        aliases,
                        rationale);

                JOptionPane.showMessageDialog(parent, "Thank You");

                setVisible(false);
                if (mf2.currentScreen != null) mf2.currentScreen.doClick();
            } else { // user closed dialog or clicked cancel
                setVisible(false);
            }
        }
    }
});

pack(); // prepare dialog for display
public void setVar(String name,
        String value,
        String units,
        String valid_range,
        String author,
        String aliases,
        String rationale) {

    // var.setName(name);
    // var.setValue(value);
    var.setUnits(units);
    var.setValidRange(valid_range);
    var.setAuthor(author);
    var.setAliases(aliases);
    var.setRationale(rationale);
}
package SSPARC;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;
import java.lang.*;
import java.util.*;
import java.awt.*;

public class SsparcTable extends TableColumnModelListener{
    private JPanel topPanel;
    private JTable table;
    private JScrollPane scrollPane;
    String columnNames[];
    String dataValues[][];
    private ClientIO cio;

    public SsparcTable(Vector columnNamesV, ClientIO c)
    {
        super();
        cio = c;
        setTitle("NA2 View");
        columnNames = new String[columnNamesV.size()+1];
        columnNames[0] = "Module";
        dataValues = new String[columnNames.length-1][columnNames.length];
        for(int i=1; i<columnNames.length; i++)
            columnNames[i]=(String) columnNamesV.elementAt(i-1);
        getTable();
        topPanel = new JPanel();
        topPanel.setLayout(new BorderLayout());
        getContentPane().add(topPanel);
        scrollPane = table.createScrollPaneForTable(table);
        topPanel.add(scrollPane);
        setSize(columnNames.length*50, columnNames.length*20);
    }

    private void getTable()
    {
        for (int i=1; i<columnNames.length; i++)
        {
            dataValues[i-1][0]= columnNames[i];
        }
    }
}

10.2.6 SsparcTable

Written by Vida Hu [23]. Modified by Quincy Scott and Tara Sainath [31].
for (int i=1; i<columnNames.length; i++) {
    for (int j=1; j<columnNames.length; j++) {
        if(i==j)
            dataValues[i-1][j] = "same";
        else
            dataValues[i-1][j] = determineRelation(columnNames[i],
                                                  columnNames[j]);
    }
}

Table = new JTable(dataValues, columnNames);
// table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);

// add a column listener to handle moving the columns
TableColumnModel columns = table.getColumnModel();
columns.addColumnModelListener(this);

for(int i=0; i<columns.getColumnCount(); i++) {
    TableColumn current = columns.getColumn(i);
    current.setPreferredWidth((columnNames[i].length()*10)+8);
}

private void changeTable() {
    for (int i=1; i<columnNames.length; i++) {
        dataValues[i-1][0] = columnNames[i];
    }

    TableColumnModel columns = table.getColumnModel();

    for (int i=1; i<columnNames.length; i++) {
        for (int j=1; j<columnNames.length; j++) {
            int current = columns.getColumn(j).getColumnIndex();

            if(i==j)
                dataValues[i-1][current] = "same";
            else
                dataValues[i-1][current] = determineRelation(columnNames[i],
                                                            columnNames[j]);
        }
    }
}

private String determineRelation(String name1, String name2) {
    Function function = cio.getFunction(name1);

    if(name2.toLowerCase().equals("constants")) {
        InputList il = function.getCurrentInputs();
        if(il.contains(Functions.CONSTANTS_CODE))
            return "X";
    }
} else {
    return "";
}

} else if(name2.toLowerCase().equals("design")) {
    InputList il = func1.getCurrentInputs();
    if(il.contains(Constants.DESIGN_CODE)) {
        return "X";
    } else {
        return "";
    }
}

} else {
    CallList cll = func1.getCurrentCalled();

    if(cll.contains(name2)) {
        return "X";
    } else {
        return "";
    }
}

public void columnAdded(TableColumnModelEvent event) {}  
public void columnRemoved(TableColumnModelEvent event) {}  
public void columnMoved(TableColumnModelEvent event) {
    int from = event.getFromIndex();
    int to = event.getToIndex();
    if(from!=to) {
        String moving = columnNames[from];
        columnNames[from]=columnNames[to];
        columnNames[to]=moving;

        changeTable();
        repaint();
    }
}

public void columnMarginChanged(ChangeEvent event) {}  
public void columnSelectionChanged(ListSelectionEvent event) {}  
   
}