VAMP EXPLORER: AN INTERACTIVE FRAMEWORK FOR NAVIGATING A COMPLEX HIERARCHY OF PVS THEOREMS

Approved by:

_____

Dr. Peter-Michael Seidel

_____

Dr. Margaret H. Dunham

_____

Dr. David Matula

# VAMP EXPLORER: AN INTERACTIVE FRAMEWORK FOR NAVIGATING A COMPLEX HIERARCHY OF PVS THEOREMS

A Thesis Presented to the Graduate Faculty of the

School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Master of Science

with a

Major in Computer Science

by

Nathaniel E. Ayewah

(B.S., Southern Methodist University, 2003)

July 28, 2005

Ayewah, Nathaniel E. B.S., Southern Methodist University, 2003

VAMP Explorer: An Interactive Framework for Navigating a Complex Hierarchy of PVS Theorems

Advisor: Professor Peter-Michael Seidel

Master of Science degree conferred July 28, 2005

Thesis completed July 27, 2005

The VAMP (Verified Architecture Microprocessor) is a pipelined microprocessor that is being verified using PVS (Prototype Verification System), a semi-automatic theorem prover. Almost 450 theories formally describe and verify all the VAMP components down to the gate level. This detail makes it possible to automatically generate an implementation from the formal descriptions but also adds complexity to the hierarchy of theories. These theories, composed of numerous lemmas and definitions, currently exist as a collection of files and directories. They are supplemented by PVS proofs, implementations in verilog and numerous publications explaining the work that has been done.

The goal of our project is to provide a cohesive interface – VAMP Explorer – that will make it easier to navigate and understand the correctness proofs of the VAMP. Our contributions include a dynamic hierarchical graph for exploring the VAMP, a schematic view for relating the theories to a traditional understanding of a microprocessor, and a

local view for identifying the properties and relationships in each theory. The VAMP Explorer is web-based to make the VAMP theories widely accessible. It is built in a modular structure so that it can adapt to changes and grow with future extensions to the VAMP.

This project supplements the existing formal descriptions of the VAMP with high level structures that present a profile of the VAMP and expose the relationships between its components. The visualization of the VAMP can be seen as a case study to demonstrate the benefits of our framework. Ultimately, our goal is to provide methodologies that can be used more generally for the visualization of complex hierarchical proofs and systems.

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## ACKNOWLEDGEMENTS

I want to thank Dr. Peter-Michael Seidel for introducing me to and steering me through this project. Many thanks also to Nikhil Kikkeri for providing initial expertise on the VAMP system and PVS in general – I'm glad I could draw from your well of experience. I would also like to acknowledge the VAMP team at Saarburg University in Germany led by Dr W.J. Paul that created the original theories and made them available to us.

I am grateful to the committee of professors at SMU that reviewed this thesis. I also got ideas or resources from individuals not directly related to the project: Dr. Joseph Kiniry from the University College Dublin exposed the challenges of parsing PVS and Devin Cook from California State University created an excellent engine for building context-free grammars.

Finally I am thankful for my family and their unending patience when it looked like I would never finish this project.

Chapter 1

# INTRODUCTION

## 1.1. General Problem Statement

The VAMP Project was started at Saarland University, Germany as an effort to formally verify a microprocessor [BB+02]. The project resulted in a large body of theorems which describe the microprocessor design and show that it is correct and bug free. The theorems were written using a theorem prover called PVS (Prototype Verification System) [OSR01]. In principle, we should be able to take these verified designs and implement them in silicon, confident of their correctness, without needing to simulate or *validate* them. Of course, synthesis and fabrication errors could lead to faulty chips but we are confident in the underlying design … or are we?

Anyone introduced to the project and considering an implementation will want to review the theories to gain a level of confidence in the quality and correctness of the work. This is a complex proposition, since reading such a large body of PVS theories is like trying to understand a large Java project by reading the Java code. The theories are written in a modular fashion meaning that a user may have to navigate through several files to find all the definitions, lemmas, or proofs that are connected to a given component. A number of papers and publications describe the paper and pencil proofs that drive the theories but do not include all the PVS theories in detail.

All of this motivates our interest in a framework for visualizing and navigating the hierarchies created by collections of PVS theories. We want to give theorem writers and readers the big picture that shows how each component contributes to the final correctness, while also allowing them to navigate through a sequence of relationships between subcomponents and theories.

Since these collections can be quite large, we should generate the final interface automatically from an existing anthology of PVS files. To do this, we should take advantage of the structured nature of PVS theories. And since VAMP describes a hardware system, we can provide an intuitive *schematic* presentation and provide a way for users to compare the structural specifications with the behavioral statements they are supposed to implement.

It would also be useful to implement an in-code commenting system like Javadoc that will support automatic documentation generation [Sun05]. Unfortunately, the large body of theories in the VAMP has already been written and does not include such structured in-code comments, so we do not address such facilities in this project.

### 1.2. Research Goals and Contributions

The goal of this project is to create an interface to the VAMP theories that can be used in general for visualizing and navigating PVS theories. Specifically, we create a system that automatically generates this interface from the existing files. We also aim to create an interface that is accessible over the web. This visualization should facilitate efforts to communicate about the theories and give readers confidence in the correctness of the theories.

We also aim to make it possible for other researchers to extend or customize the visualizations. To this end, we separate the project into two tasks. The first effort is to extract and structure the information in the theories. These include the lemmas and definitions as well as related proofs and theories. We format this extracted structure using XML. This uniform structure drives the second stage of the project which is to visualize the information using a web-based interactive interface.

This project has also created other useful contributions. We present a schema for representing PVS theories in XML particularly when the goal is to visualize the theories. Our research has also led to a simple method for generating schematic representations of PVS constructs that are flagged as hardware structures. To facilitate these methods, we have created a parsing engine for PVS files.

## 1.3. Background and Related Problems

### 1.3.1. Formal Verification

Traditionally, engineering fields place a great deal of emphasis on correctness in design. This focus is necessary because faults in engineering design can cause great inconvenience or even have fatal consequences. Computer hardware and software engineering have often been criticized for not adopting this same rigor. Engineering correct hardware and software designs is challenging because it is intractable to simulate every possible input case especially for complex designs. Furthermore, market pressures often lead to short design cycles [KG99]. In general, it is not always possible to completely guarantee the correctness of a design because we cannot completely formalize the intentions of the designer. However, we can use a number of existing approaches to

check certain properties of the design and guarantee that it is equivalent to a given formal specification.

Often designs are validated using simulation to predict the output given a set of input vectors. While this kind of validation is not usually exhaustive enough to give us confidence in our design, it can often identify most of the errors [Kr99]. On the other hand, formal verification provides proof that a design or implementation meets a set of requirements or exhibits certain desired properties. Techniques for formal verification range from fully automatic approaches such as model-checking and equivalence-checking to semi-automatic interactive theorem proving. The automatic methods are currently more popular because they are less complex. But they are also less expressive and do not scale well to large or complex circuits.

### 1.3.2. The VAMP Perspective

Interactive theorem proving has not been completely accepted for verifying large designs because it is thought to be too time consuming and expensive, requiring considerable expertise. The VAMP project set out to explore the feasibility of using theorem proving to verify a microprocessor, complete with a pipelined architecture, an out-of-order scheduler, a cache memory interface and complex arithmetic units. It uses PVS, an interactive theorem prover, to specify the structure of the different components and their intended behavior and to prove that this behavior is accomplished by the structure. PVS supports specification down to the bit-level which makes it possible to automatically generate a fully functional implementation (in Verilog) from the given theorems.

### 1.3.3. The Visualization Problem

The VAMP project has demonstrated that verification of such complex systems can be accomplished in reasonable time and at reasonable cost. However the theorems do nothing to hide the complexity of the system – indeed they exacerbate the problem by bringing in the nuances of the PVS specification language. Currently, the VAMP collection of theories have been arranged in files in a directory structure and made available over the web [VMP05]. In addition a number of publications have been created to describe the work that has been done. But a user wanting access to the theorems still has a hard time navigating through them and inspecting the proofs that have been applied to them. Figure 1.1 shows a hierarchy of some of the theorems in the VAMP system and illustrates some of the challenges associated with navigating the VAMP:

- The hierarchy is very deep and provides detailed representations even at its lowest levels.
- There are many interconnections between nodes.
- The high level components verified in the VAMP are related across interfaces that sometimes span multiple theorems.

Our goal is to create a VAMP explorer that will give users new ways to visualize the hierarchy and navigate through it. We believe this will make the VAMP theorems more accessible to researchers and students wishing to better understand or use it.

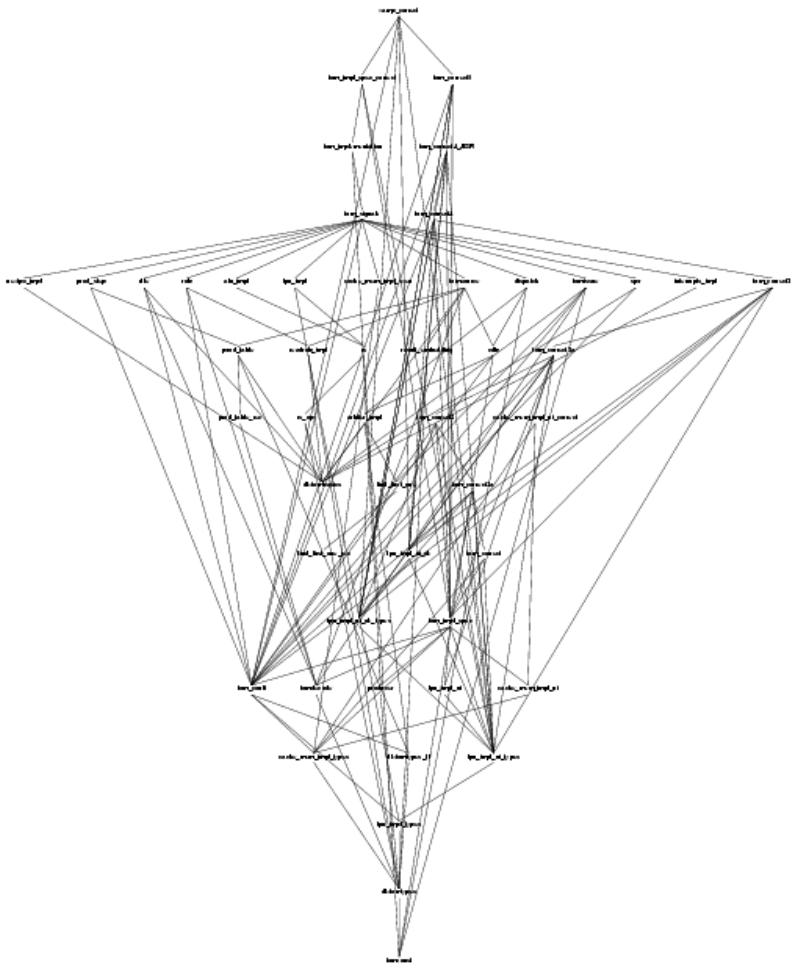


Figure 1.1. A Subset of the Relationships between VAMP Theories

## 1.4. Related Work

### 1.4.1. Verification of Hardware Designs

The VAMP is described in a collection of publications and theses written at different stages of the verification project. Much of the initial work is done using paper-and-pencil proofs in [MP00] which includes descriptions for a pipelined DLX-based implementation with an IEEE compliant floating point unit and interrupts. [BJK01] describes the PVS proofs used to verify some of the basic circuits such as “incrementers, adders, arithmetic

units, multipliers, leading zero counters, shifters, and decoders." [Krö01] verifies the pipelined architecture and scheduling algorithms, [Ja02], [BJ01] and [Ber01] describe the floating point unit and [Be04] describes the verification of the cache memory interface. The architecture is based on the DLX instructions set described by Hennessy and Patterson [HP96] and the out-of-order scheduling is based on Tomasulo's algorithm [Tom67].

Other work has been done to formally verify microprocessors and other hardware designs. [KG99] and [Kr99] describe advances and techniques in model checking, equivalence checking and theorem proving. Hardware designs have also been verified using other theorem provers including HOL [Fo01] and ACL2 [BKM96].

### 1.4.2. Visualization of Hierarchical Structures

Visualization is the "process of transforming information into a visual form enabling the user to observe the information" [GCE99]. Information visualization has many applications in data mining, data management, networking and other fields. Visualization can make information more understandable and support creative modeling [CW+00]. It can be used to analyze data and uncover trends [WB98] or enhance a user interface.

Hierarchical visualization shows relationships between data items arranged in a tree structure. Hierarchies are often shown using directed acyclic graphs and treemaps. Treemaps are a space constrained visualization that show the highest levels of the hierarchy and allow a user to zoom down [BSW02]. Research in hierarchical visualization is conducted by groups wanting to visualize XML, databases or other structured relationships. Some recent advances include an enterprise knowledge platform

for delivering information in a hierarchy within a given context [Bra05] and methods for dynamically changing the visualizing of XML according to what a user wants to capture [JS04].

The PVS interactive development environment (IDE) focuses on providing an effective interface for theorem development and not on presenting and navigating completed theorems. As a result, past research in visualization of PVS has focused on providing functionality that is popular in modern IDEs such as built in contextual help, quick access to definitions, code completion, and pretty printing [Kin03].

### 1.5. Organization

In chapter 2, we describe the basic ideas that form the theoretical foundation for our work including principles in theorem proving, visualization and user interface design. We then go on to do a detailed requirements analysis in chapter 3 to determine and prioritize the features that should be included in the VAMP Explorer. Chapter 4 describes our effort to extract structured data from the existing bed of files and chapter 5 describes our visualization approaches. In chapter 6 we discuss issues related to system deployment, both for users wanting to visualize theories and researchers wanting to extract structured data from their projects. Finally, we conclude in chapter 7 by describing requirements we were not able to address and possible future directions for this project.

Chapter 2

# BASIC CONCEPTS AND TERMS

## 2.1. Formal Verification and Theorem Proving

Verifying the correctness of hardware and software designs is an inherently difficult problem. Most students in these fields are taught to identify or recognize test cases that represent the wide spectrum of possible conditions a system can be in. This approach, enhanced with clever simulations, continues to be employed as the students move into industry. Yet in industrial designs, these test cases and simulations often account for only a small subset of possible conditions for the system.

It has to be said that many successful designs have been completed and released despite this inability to test all cases. This is because all designs depend on some *informal* logical reasoning done by a human designer [PS05]. The effectiveness of this reasoning depends on the complexity of the problem and the experience of the designer. Formal verification aims to improve the application of this reasoning by making it more precise and rigorous. Unfortunately some abstractions in the human reasoning process are difficult to express using the most accessible formal verification methods. Theorem proving provides the greatest potential for overcoming these difficulties.

### 2.1.1. Formal Verification Methods

Formal verification methods include equivalence checking, property checking and theorem proving. The goal of equivalence checking is to prove that two different specifications of a given system are functionally equivalent to each other. This is useful when a specification (such as a high-level design) already exists and a new lower-level implementation is to be created. Hence the correctness of the new system assumes that the original specification is correct. The advantage of equivalence checking is that it can be done automatically using techniques such as binary decision diagram (BDD) equivalence and state space traversal. The disadvantage is that it is not very expressive and is only feasible for relatively small circuits [Kr99].

Property checking approaches model the system and certain properties using propositional temporal logics. These approaches show that the desired properties are always true using symbolic state traversal techniques. Again, these approaches suffer from a lack of expressiveness for dealing with large complex circuits [Kr99].

### 2.1.2. Theorem Provers

In theorem proving, the goal is to show through mathematical reasoning that a given implementation is equivalent to a given specification. The specification and implementation can be abstractly expressed using formal logics. These logics range from First-Order logics which are decidable and hence support some automation to higher order logics which require interaction from a human to complete the proofs.

Theorem provers are very expressive, especially when using higher order logic, and can be used to describe and verify large and complex circuits. Still, they are not popular

because the proofs cannot be written automatically. Instead, the theorems created must be proved interactively which can be a time consuming process.

## 2.2. PVS Concepts

PVS is a research prototype developed at SRI International for writing higher order logic specifications and interactively proving them. It includes a specification language, an extendible theorem proving language and some predefined theories. It is implemented in Common Lisp and uses the popular Emacs editor as its interface.

At the highest level, a PVS specification is either a *theory* or a *datatype*. These specifications can be linked using *import* and *export* statements [OSRS01]. A PVS specification file (extension '.pvs') can have one or more theories or datatypes. Each specification file has an associated proof file (extension '.prf') that holds the lisp proof scripts produced during the last interactive theorem proving session. Several related PVS files are grouped into a *context* which usually corresponds to the directory the files are in.

PVS theories consist of type and constant declarations, expressions involving types, constants, variables and/or functions, and formulae which state some assumption or hypothesis. Completed specifications must first be parsed to ensure syntactic consistency. Then they need to be *typechecked* to ensure semantic consistency. PVS' rich type system allows the creation of complex types (including functional types). This makes the problem of checking the consistency of the type system undecidable. In some cases an interactive theorem proving step is needed to prove *type-correctness conditions* (TCCs) before the typechecking phase is complete.

Once specifications have been parsed and typechecked, the interactive theorem prover can be fired up to prove the lemmas or theorems in the specification. The prover provides powerful commands that can reduce, rewrite or simplify formulas. It also provides *proof stategies* for doing induction, recursion or other useful activities. PVS supports the creation of user-defined proof strategies which combine commands in different forms and can be used to represent a paradigm for approaching a class of problems.

The PVS system includes a set of predefined theories called the *Prelude*. These theories form the foundation for the theorem prover and define basic concepts that can be used when writing specifications. These include theories that describe "logic, functions, relations, induction, sets, numbers, sequences, sum types, quotient types, and mu-calculus" [OS03].

### 2.3. Visualization Concepts

Modern technological advancements like the World Wide Web, email and personal digital assistants have contributed to an explosion of information. Deriving useful knowledge from all this information is a growing challenge. Research in visualization aims to take advantage of human cognitive abilities to present information. This has led to the creation of tools like tree maps, fish-eye lens-based components, tool tips, and zoomable interfaces [Vis05].

### 2.3.1. Human Cognition

Cognition refers to our mental abilities (or limitations) as human beings. In visualization, we are concerned with cognitive processes that include memory, perception and recognition, attention, learning and problem solving [SPR02].

For example, the way information is structured can affect our ability to focus on salient features. It is easier to read the contents of the database when they are structured using a table format, than it is to read them in a paragraph format. Another example involves the use of familiar icons to represent certain actions. The choice of pictures in these icons determines how well the user can learn to use the interface. Throughout our project, we consider these principles as we decide what widgets to use in our interface.

### 2.3.2. Visualizing Hierarchies

Hierarchies are created when data is structured in a tree-like format with groups of data forming subtrees. Hierarchies can be visualized using a directed acyclic graph (DAG) which visually groups related parent- and child-nodes and visualizes this relationship with an edge between the nodes. Variations on the DAG include tools that zoom into a particular node or hide nodes that are not in use.

A treemap is another effective way of visualizing hierarchies. It constrains the entire hierarchy to a fixed space and uses boxes to represent the highest level nodes. The subtree of a node is represented by smaller boxes contained in the node. The treemap also uses color as another dimension. For example, the color intensity of a box may indicate the magnitude of a value in its subtree.

Some of these visualization components are shown in Figure 2.1.

**TreeMap**


**Newsmap** uses different colors to represent categories of news and the size and intensity of the boxes to reflect the popularity of the news item.

Source: http://www.marumushi.com/apps/newsmap/newsmap.cfm (Accessed July 13, 2005)

**Directed Acyclic Graph**

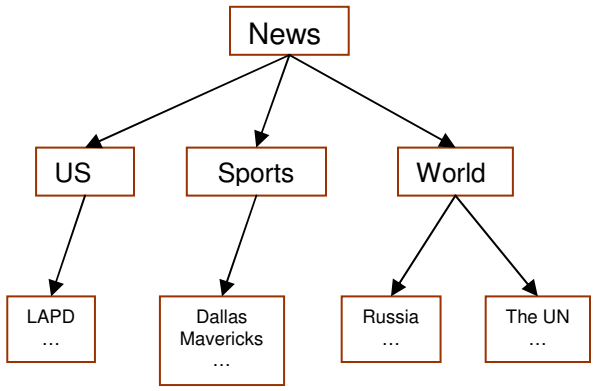



Figure 2.1. Hierarchical Visualization Components

## 2.4. User Interface Design Concepts

The best visual interfaces are often those that allow a user to directly manipulate them. In this section, we describe a small subset of interaction design principles that, when applied, lead to the most effective and user-friendly interfaces. Some of these principles relate to the contents of a good interface, but there are also important principles regarding the methodology we use to generate these interfaces.

### 2.4.1. Usability Goals

Most users have had the experience of interacting with an interface that was difficult to use or frustrating. Perhaps it did not do what they expected in response to their actions,

or perhaps it was difficult to learn. As we design our interface, our goal should be to optimize the user's interaction and their ability to complete their tasks. Some specific goals are that the interface is effective, efficient, safe, useful, learnable and contains features that are easy to remember [SPR02].

### 2.4.2. User Experience Goals

In interaction design, our focus should be on the user we are designing for, and on ensuring their satisfaction. The user's experience will impact the usefulness of the interface to them. Some user experience goals include making the interface enjoyable, helpful, supportive of creativity, and aesthetically pleasing [SPR02].

In chapter 3, we analyze our users and determine what goals we should have as we design the interface.

Chapter 3

# REQUIREMENTS ANALYSIS AND CONCEPTUAL DESIGN

## 3.1. User Analysis

In this section, we wish to identify all the potential users of our system so that we can direct our design to best support the needs of these users.

### 3.1.1. Primary Users

These users include researchers using PVS or interested in the VAMP. They represent the individuals that will interact with our system the most. Specifically:

*PVS Researchers* may want to use the system to visualize their own theories but they may not be familiar with web/visualization concepts, or data concepts such as XML. They also may not be familiar with the hardware concepts used in the VAMP.

*PVS Readers* include those seeking to study a body of theorems previously constructed using PVS. They may not be familiar with PVS constructs and other concepts associated with our system.

*VAMP Readers* include those seeking to learn about the VAMP. Some of these users may not care about PVS-specific details, but want to know how the components of the VAMP come together to show correctness of the system.

### 3.1.2. Secondary Users

These include users that may interact with our system but do not need to comprehend all aspects of the system to meet their goals. This could include technical managers trying to evaluate different formal verification techniques and wanting to experience theorem proving at work on a full scale system like the VAMP. Other potential users are formal verification engineers using other techniques such as property checking or equivalence checking. These users will benefit from the interactive nature of our system but are not our primary users. So, for example, our final interface will not devote many resources to explaining theorem proving.

Finally our system could impact users in society who use products that have been verified using PVS because this system could contribute to the correctness or acceptance of the theorems.

This understanding of the intended audience is what drives our design and choice of features in the final system. For example, we aim to use visualization structures that are intuitive since many of our users will not be familiar with visualization principles.

## 3.2. Needs Analysis

In this section, we highlight the user needs that motivate our design and guide our design decisions.

*The Big Picture*: Users and designers of large systems need to be able to see the big picture to aid understanding and development. How does an individual theorem or lemma contribute to the success of the entire system? What are the theories at the top of the

hierarchy? Which theories support different subsystems such as the memory interface or the pipelined scheduler?

*Correctness Checking*: Formal verification is necessary to ensure systems are correct and safe. Exhaustive checking is usually not possible. Other formal verification methods are not as expressive as theorem proving and cannot handle complex systems. Users need a system that will give them confidence in the correctness of the proofs and, consequently, the correctness of the design.

*Theory Understanding*: Users need help understanding individual theories. This may come in the form of paper and pencil proofs, or documentation explaining the PVS constructs used. This information is often separate from the theories. It would be very helpful to put this information at the users' fingertips as they read the theories.

*Theory Access*: Users and developers need a way to quickly find theories and to compare theories or to find individual components. They may be interested, for example, in discovering where an abstract datatype is defined or in reading the description of a lemma that is used to support the current theory.

*Theory Presentation*: After creating the theories, designers and researchers need a way to talk about it to others. Web based visualization provides a framework for public and global distribution. The theories can be further enhanced by using syntax coloring to expose keywords and identifiers.

The existing emacs-based PVS development environment meets some of the needs described above. For example it provides some syntax coloring facilities and provides

commands for visualizing a subset of the hierarchy and for visualizing proofs. But it does not have a web-based front-end nor does it provide advanced facilities for quickly navigating through a VAMP hierarchy.

## 3.3. Requirements Specification

### 3.3.1. Use Case Diagram

The Use case diagram describes some of the high level requirements of our system.

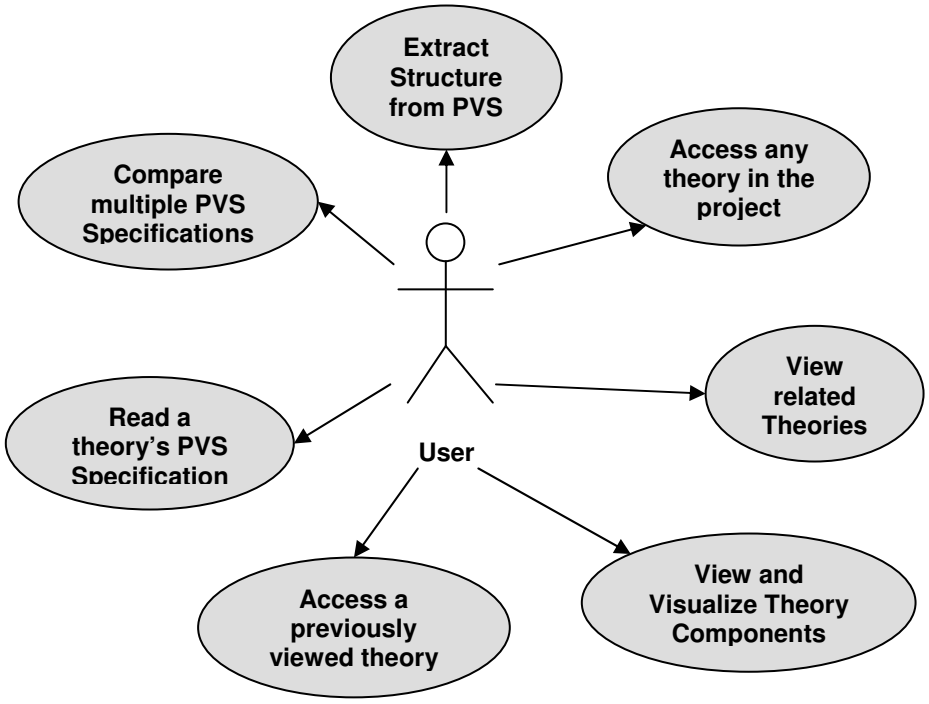



Figure 3.1. Use Case Diagram

### 3.3.2. Functional Requirements

- The system should visualize an overview of the body of theories and the relationships between them.

- The system should provide different views of the same theory that provide different kinds of information. For example, one view may show how other theories are related to the given theory; another view may show what components make up the given theory.
- The system should identify and classify subcomponents of a theory and show the relationships between components, as well as relationships to pertinent lemmas, proofs or definitions. A user should be able to navigate to these related components.
- The interface should be automatically generated from a body of pvs files (with few support files needed). The researcher should not have to do a lot of extra work to support the interface. (The exception might be the inclusion of documentation to create an automatic documentation generation system. This is not done in this project, in part because the VAMP theory files do not contain structured inline comments that support this.)
- The system should provide a split-screen to allow users to compare different code from different theories or different parts of the same theory.
- The system should provide facilities for a user to selectively hide parts of the hierarchy or focus on a subset of the hierarchy.
- The system should provide some documentation to aid the user's understanding of the theories
- The system should pretty print and/or syntax color theories to make them easier to read.

- The system should automatically identify structural components that can be represented as schematic symbols and create a (canonical) schematic view of the whole system.
- The system should provide quick access to any theory in the system.
- The system should keep track of the theories a user has explored and allow a user to traverse back and forth along this history. The feature will be familiar to users with experience browsing the web and will allow users to return to a theory after temporarily leaving it.

### 3.3.3. Usability Requirements

- Complex relationships should not obscure user's ability to see the hierarchy
- Though most users will have some expertise in hardware design, formal verification and/or theorem proving, the interface should still be intuitive using easy to understand visualization concepts.
- The system should provide enough and concise documentation so users lacking expertise in certain areas can learn to use it fairly quickly.

### 3.3.4. Data Requirements

- The system should parse the PVS format into a structured format that is readily accessible and is structured to support the tasks of the interface.
- A generic data format such as XML should be used to drive the interface so other kinds of interfaces can be generated from this structured content.

### 3.3.5. Time Requirements

- The interface should be very responsive even when representing bodies of theories that have a lot of content such as the VAMP. To do this as much processing as possible should be relegated to a preprocessing stage that converts the PVS content into structured interface-accessible content.

### 3.3.6. Environmental Requirements

- The system should be web-based and accessible to most web users. This means it should run in browsers like Internet Explorer and Mozilla-based browsers, and on operating systems such as Microsoft's Windows, Apple's Macintosh, and Linux. It should also have reasonable bandwidth and processor requirements.
- The preprocessing function that automatically extracts structured data content from the PVS files should be accessible on a wide range of platforms. This is because the PVS files may originally reside on different operating systems such as Unix, Linux and Windows.

## 3.4. Usability Goals

In this section, we describe some general usability goals that we aim to keep in focus as we develop our interface. Some of these goals are applicable to most user interfaces and applying them leads to more user friendly interfaces [SPR02].

- A user should be able to quickly *recognize* the tools or commands to perform desired tasks instead of having to *recall* them or refer to help documentation.

- Our design emphasis should be on making the interface useful to the user rather than providing many features.
- There should be few unexpected features and these should be easy to understand.
- The explorer should provide a consistent layout even as users use different methods to visualize their content.
- The explorer should prevent the user from providing erroneous inputs by using buttons or dropdown menus for predefined inputs.
- The user should have the freedom to move between different visualizations while exploring a theory.

## 3.5. Preliminary Designs and Prototypes

While identifying the requirements for the VAMP Explorer, we developed some initial designs and built some proof-of-concept prototypes. These exercises revealed some significant challenges that we would need to overcome to complete this project.

### 3.5.1. Preliminary Screenshots

Figure 3.2 shows our initial concept. Here we only use a manually extracted subset of the VAMP system but users already have access to a directed acyclic graph showing the import relationships between theories. The theories come from different contexts and are color-coded to reflect this. Each theory is labeled: *context@theory*, which is the convention that occurs in PVS import lists. The interface provides tabs for switching between different views of the relevant theories. In the lower window, we display different kinds of content relevant to the current theory.

In Figure 3.3, we explore a method for automatically generating the directed acyclic graph. We generate and layout our graph using Graphviz, a popular graphing application [Gvz05]. This exercise leads us to identify three steps that are needed to visualize the hierarchy of theories.

1. Identify the relationships between the theories. In this prototype, this task was completed manually, but the final system will need to extract this information dynamically using the import lists in the theories.
2. Create and layout a graph based on these relationships. The layout algorithm should be optimized for hierarchical structures as it decides, for example, which nodes should appear at the top of the graph and which should appear at the bottom. In this case, the graphing application is used for layout only and is not a part of our final interface. Hence the graph properties (node coordinates and edges) need to be exported into a standard format such as XML.
3. Redraw the graph using the final interface. In this case, our interface built using Macromedia Flash [Mac05] and the graph is drawn using the coordinates from step 2.

In Figure 3.4, we introduce a new view for navigating the hierarchy. This *Local View* is similar to the dynamic graphs from TheBrain which visualize information in its context by showing all relevant relationships [Bra05]. In this case relevant relationships include constituent lemmas, imported theories, applicable views and 'parent' theories (i.e. theories that use the current theory). In this exercise, we discover that identifying these

parent theories may be complicated because this information isn't naturally contained in the current theory's PVS specification.

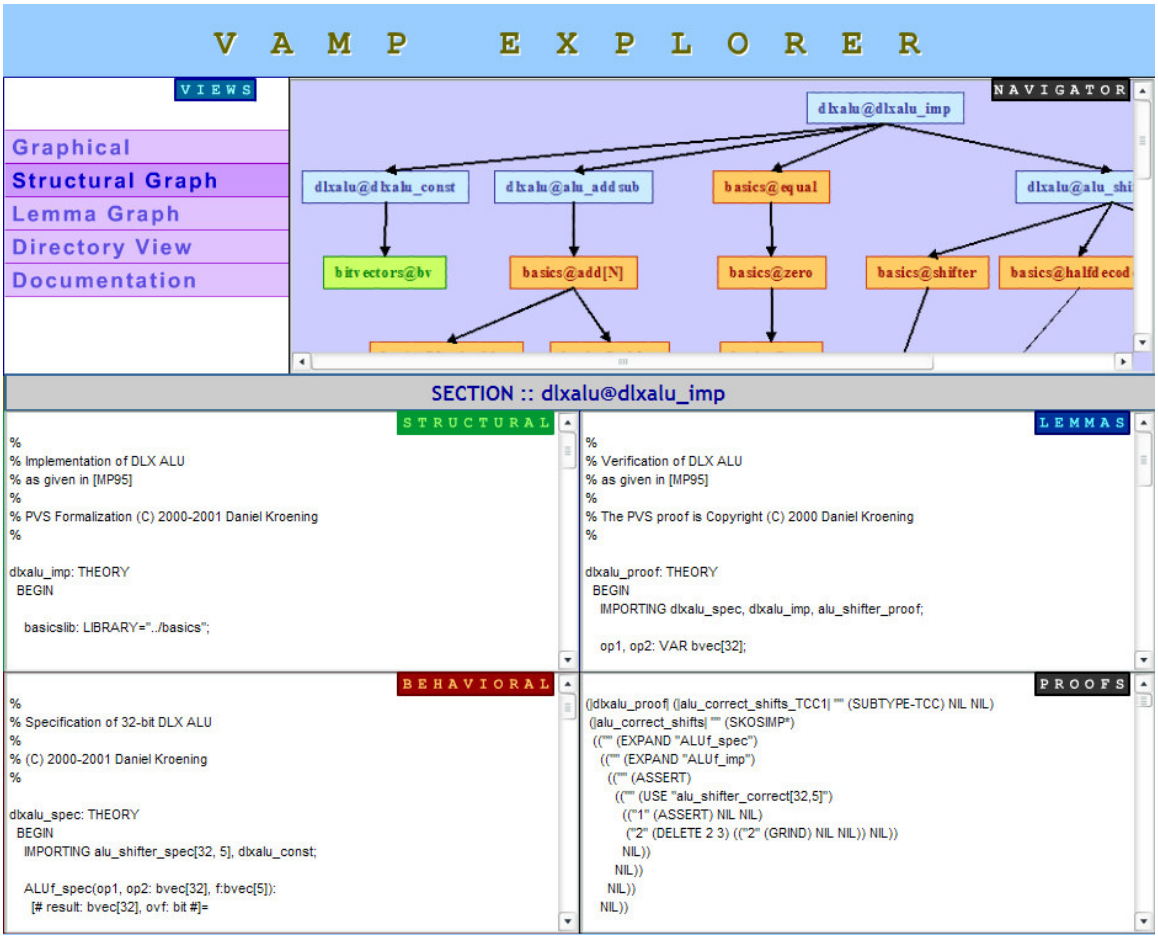



Figure 3.2. Preliminary Prototype of VAMP Explorer

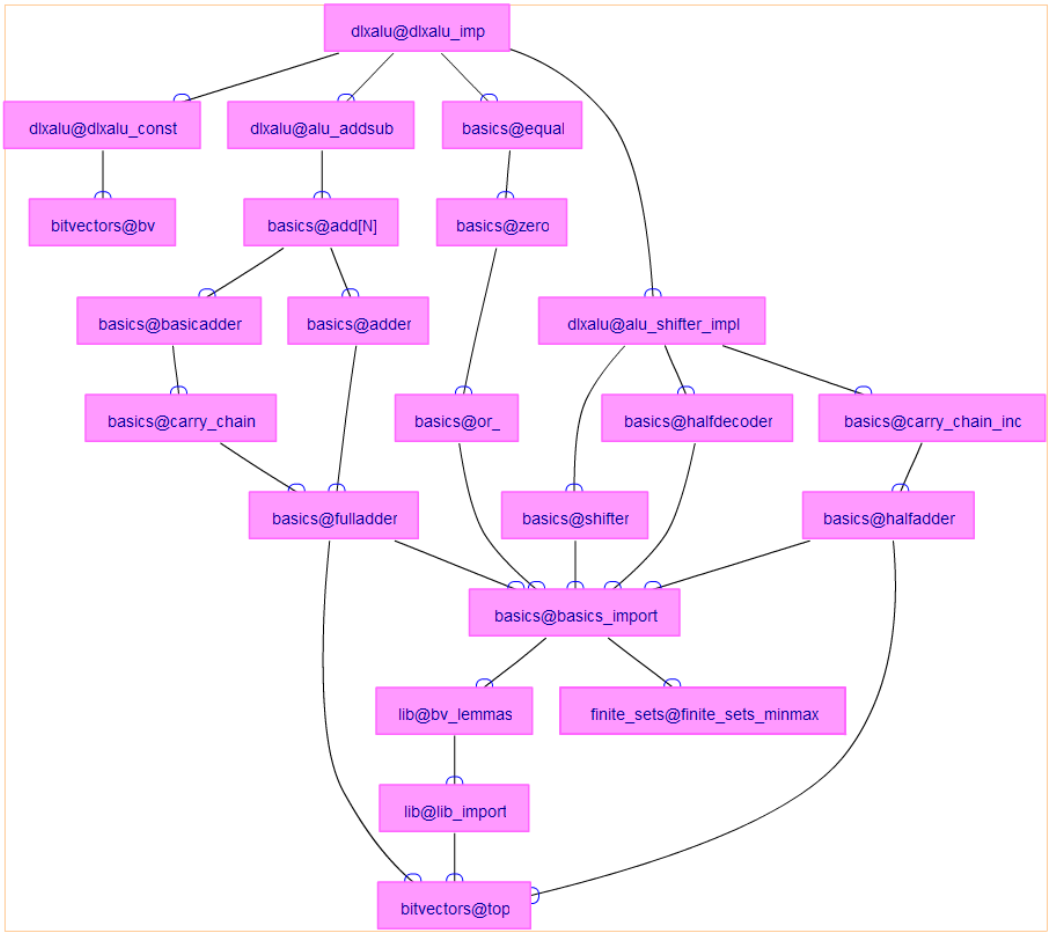



Figure 3.3. Automatically Generating a Directed Acyclic Graph

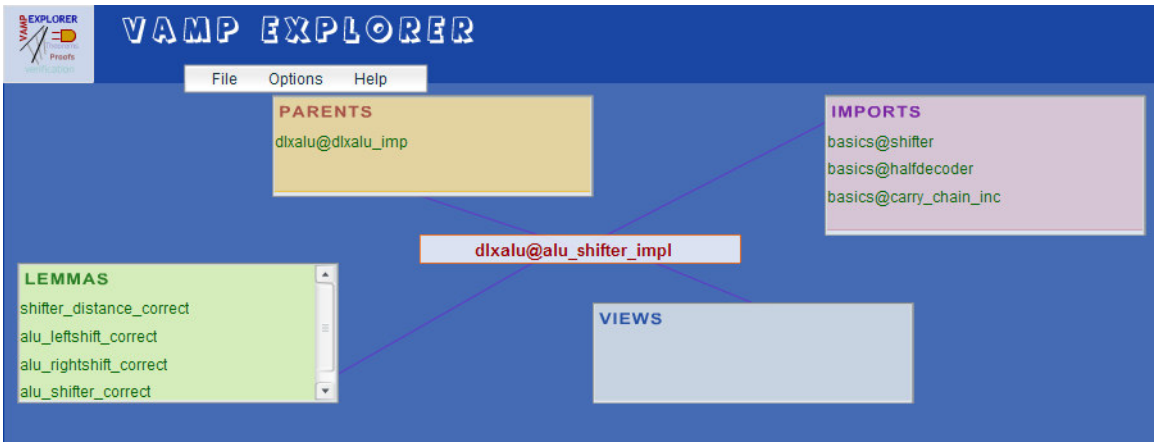



Figure 3.4. The *Local View* Concept

### 3.5.2. Early Challenges

These initial prototypes expose some problems we will need to address as we approach our system design and implementation. Some of these are:

- *Automatic Extraction:* the process of manually preparing the data files to drive these demos was time consuming and will not scale to the full VAMP system. Hence we need to extract this information automatically. This extracted content includes the relationships between theories and the relationships between subcomponents that make up theories.
- *Graph Layout:* we need to use graph layout algorithms that make efficient use of space and expose the intrinsic hierarchical nature of the relationships. Even with this small illustration, the graph generated was quite complex – visualizing the entire VAMP hierarchy will be difficult. One initial idea is to selectively hide parts of the tree. This may be complicated because it would require that the layout algorithm be rerun every time a new section of the theory is hidden or revealed.
- *Scaling the Interface:* when we interact with the full VAMP system, the interface will need to be responsive even when dealing with large files.
- *Schematic View:* we will need to represent components with an appropriate schematic gate and extract the inputs and outputs to these gates. In addition, we will need a routing algorithm to layout these gates.
- *Modeling the user's interaction:* In these prototypes, we use different paradigms to allow a user to navigate between different views and different widgets to

display the PVS code. We will need to identify the paradigms and components we want to use in the final interface.

## 3.6. High Level and Conceptual Design

Figures 3.5-3.7 introduce a high level design detailing the implementation tasks relevant to each major activity performed in the system. At the highest level, the tasks *Extract PVS Structure* and *Visualize VAMP* represent the two major phases in this project. Chapter 4 is devoted to a detailed description of the design and implementation used to extract the PVS structure, while chapter 5 describes the visualization effort.

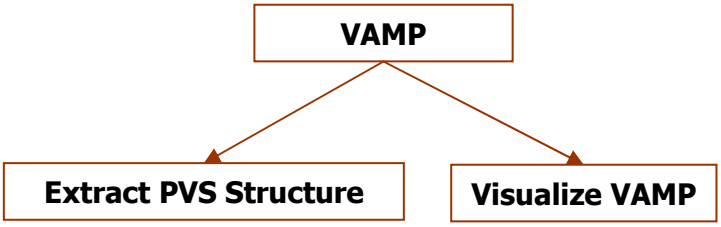



Figure 3.5. High-level Design: Overview

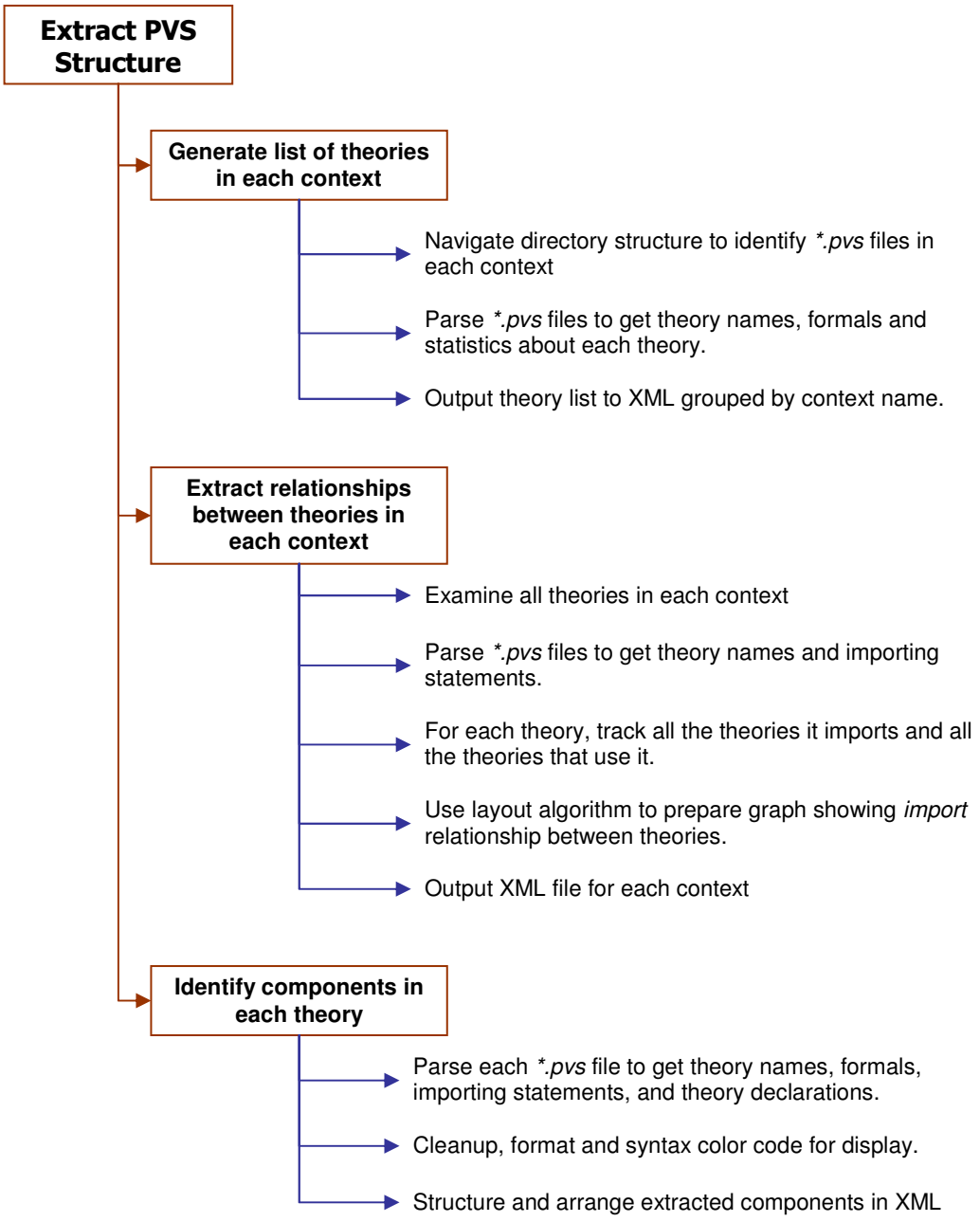



Figure 3.6. High-level Design: Data Extraction Phase

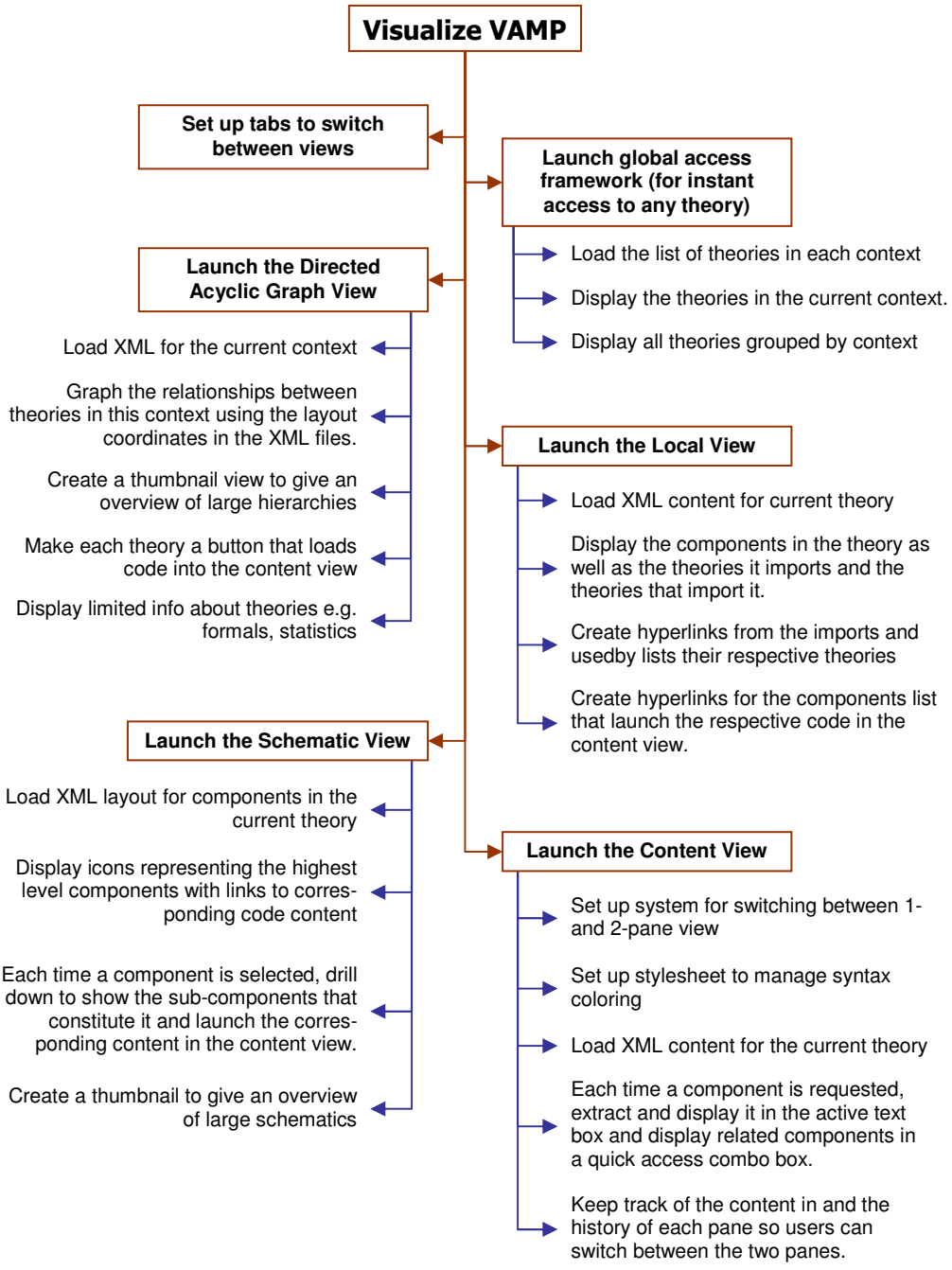



Figure 3.7. High-level Design: Visualization Phase

Chapter 4

# EXTRACTING STRUCTURED PVS CONTENT

## 4.1. Parsing PVS

In this chapter, we describe the considerable effort to extract information from the existing bed of PVS files in the VAMP and to structure this information to support our visualization requirements. Our initial attempts led us to experiment with some of the *status* and *display* commands in PVS [OSR01]. The status commands indicate which related theories or formulas have been typechecked or proved. A side effect of this is that they expose some of the relationships between components and theories. The display commands can be used to generate a graphical representation of the theory hierarchies and proof trees.

Unfortunately, the information provided from these built in commands is not sufficient nor is it structured to maximize flexible access to the content for visualization. At a basic level, the output of these commands is not formatted using a structured format like XML. In addition, these commands do not provide detailed information about the individual theory components and how they should be classified.

This leads us to consider building a parser for PVS. The PVS installation provides a complete grammar for the PVS specification language in extended Backus-Naur Form (BNF) [OSR01b]. Unfortunately, the language is very context sensitive and will need to

be modified for our purposes. The details of these modifications are described in section 4.1.3. Our choice of parsing development engine is influenced by this need for flexibility.

### 4.1.1. The Grammar Oriented Language Developer (GOLD)

Usually parsers are built using *compiler-compilers* such as YACC and ANTLR which incorporate source code describing the parser's actions with the grammar describing the language. The compiler-compiler uses this information to create a parsing program which can then be used to parse source files in the specified language. The disadvantage of this is that the grammar specification and source code are language dependent and tightly coupled – changes in the grammar could lead to significant rewriting of the source code.

The GOLD parsing system was created by Devin Cook at California State University. Its basic paradigm is to separate the task of specifying the grammar used in a parser program from the actions associated with tokens and rules in this grammar. This has the advantage of making it possible to build parsers in a language and platform independent way [Co04]. It also gives us the flexibility to focus on the grammar, making modifications where necessary, without considering the source code actions until we are satisfied with the grammar.

The GOLD system is used in two phases. The first phase uses a LALR (Lookahead Left-to-right Right-derivation) algorithm to compile a BNF grammar into a table that represents every possible state the parser can be in as it processes the text the grammar represents. During parsing, each time a new token is received by the parser, it must decide based on the states in the table whether to *shift* the token onto a stack containing other tokens that will eventually make up a rule, or to first *reduce* all the tokens on the

stack to one rule before shifting the next token. The GOLD system features a builder that is used for compiling BNF grammars into LALR tables and testing the grammars with sample text to ensure that it is not ambiguous. This phase is completely language independent and can be completed before moving to the second phase.

In the second phase, a language specific *engine* is used in conjunction with the LALR table to parse a source text and perform desired actions. In this project, we choose a Java engine written by Matthew Hawkins and featured on the GOLD Parser website [Co04]. With this engine, a simple loop can be used to specify the appropriate action each time a new token is received or a rule is reduced.

### 4.1.2. The PVS Specification Language

In this section, we highlight some of the key features of the PVS grammar that we exploit to create our structured XML representations of the VAMP files. The full grammar is presented in [OSR01b]. The grammar is specified in extended Backus-Naur Form which provides shorthand notation for dealing with common occurrences such as repeated tokens, lists, optional tokens and alternate tokens. Unfortunately, the GOLD system currently only works with standard BNF, so the PVS specification must be written out in this form.

Figure 4.1 shows some of the important rules in the grammar. The start symbol points to the highest rule in the grammar – the rule that represents a complete and correct input. In this case the start symbol – *Specification* – represents an entire PVS specification file which can contain one or more theories or datatypes. A theory is created using a shell of keywords interspaced with several optional parts (indicated with square brackets). The

most important part to us is the *TheoryPart* which contains one or more *TheoryElement*s optionally separated by a semicolon. It is these theory elements that form the basic components of the VAMP system. Theory elements include import lists, formulas, type and variable declarations, and library statements. In section 4.2, we structure the content of PVS file for visualization by isolating each of these theory elements and grouping similar ones.

```
"Start Symbol" = Specification

Specification ::= {Theory | Datatype}+

Theory ::= Id [TheoryFormals] ':' 'THEORY'
        [Exporting]
        'BEGIN'
             [AssumingPart]
             [TheoryPart]
        'END' Id

TheoryPart ::= {TheoryElement [';']}+

TheoryElement ::= Importing | TheoryDecl

Importing ::= 'IMPORTING' TheoryNames

TheoryDecl ::= LibDecl
             | TheoryAbbrDecl
             | TypeDecl
             | VarDecl
             | ConstDecl
             | RecursiveDecl
             | InductiveDecl
             | FormulaDecl
             | Judgement
             | Conversion
             | InlineDatatype

AssumingPart ::= 'ASSUMING' {AssumingElement [';']}+ 'ENDASSUMING'

AssumingElement ::= Importing
                  | Assumption
                  | TheoryDecl
```

Figure 4.1. A Subset of Rules in the PVS Grammar

### 4.1.3. Dealing with Shift-Reduce and Reduce-Reduce Conflicts

One problem that becomes immediately apparent when we try to compile the given PVS grammar is that, because of the context-sensitive nature of PVS, many shift-reduce and reduce-reduce conflicts occur. A shift-reduce conflict occurs when the LALR algorithm cannot decide whether to shift the next token onto the stack as part of a larger rule, or to reduce the tokens on the stack to a rule. A reduce-reduce conflict occurs when the tokens on the stack can be reduced to more than one rule.

Figure 4.2 demonstrates these conflicts in the PVS grammar. When the parser is reading one of the theory elements in Figure 4.1, it does not know which one it is reading since it only looks one token ahead. When it sees a string of comma-separated ids, it cannot tell if it is reading a *LibDecl* or a *VarDecl*. In other words, it does not know the *context* in which to interpret the ids. In the *LibDecl* context, these ids should be reduced to the *Ids* rule, while in the *VarDecl* context they should be reduced to the *IdOps* rule. This is a reduce-reduce conflict. A shift-reduce conflict occurs if the parser encounters an operator (*Opsym*) in this string of ids. Now it needs to decide if it should *shift* the operator unto the stack leading to the eventual reduction of an *IdOps* rule, or to *reduce* the existing stack tokens into an *Ids* rule.

Our paradigm for dealing with this problem is to modify the grammar to make it more general. In our above example, we can accomplish this by replacing the *Ids* non-terminal in *LibDecl* with an *IdOps*. The implication of this approach is that any files that are parsed successfully by the PVS system will also be successfully parsed by our system but not vice versa. In this example, any library declarations that parse successfully in PVS

will be accepted by our custom parser because *IdOps* is more general than *Ids*, but a library declaration that includes operators and parses in our parser will not parse in PVS. This is O.K. because our system will not be used to write PVS specifications and we can assume that PVS files provided to us have already been successfully parsed in the PVS system. The modified BNF file is presented in Appendix A.

```
LibDecl ::= Ids ':' 'LIBRARY' ['='] String

VarDecl ::= IdOps ':' 'VAR' TypeExpr

IdOps ::= IdOp++','

IdOp ::= Id | Opsym

Ids ::= Id++','
```

Figure 4.2. The Shift-Reduce and Reduce-Reduce Conflicts

### 4.2. XML Schemas

In this section, we design the XML schemas used to structure the components using different paradigms depending on what view is driven from these schemas. The goal is to create schemas that structure the PVS content in the VAMP such that anyone can create a visual interface to this content. The XML content for each schema is generated using the respective steps in Figure 3.5 and this content is used to drive the views in Figure 3.6. In many cases, the <*string*> object is used to specify a display string to be used in the visualization phase.

### 4.2.1. The Context Theories Schema

Table 4.1. The Context Theories Schema

| XML Tag Name | Level | Attributes | Comments |
|---|---|---|---|
| <ContextTheories> | 1 | - | Root node |
| \| – <context> | 2 | id | One object for each context |
| \| – – <theory> | 3 | - | One for each theory |
| \| – – – <id> | 4 | - | Unique theory id |
| \| – – – <filename> | 4 | - | PVS file containing theory |
| \| – – – <theoryformals> | 4 | - | *optional*; Group all the theory's parameters. |
| \| – – – – <theoryformal> | 5 | - | |
| \| – – – – – <string> | 6 | - | The display string for this formal |
| \| – – – – – <importing> | 6 | - | *optional*; importing statement that precedes string |

This is the simplest schema. It encapsulates each context in a *context* tag and lists all the theories that occur in that context. Theory objects contain an id, the PVS file containing the theory, and an optional description of some of the formals or parameters associated with the theory. (A theory's id should be unique in its context.)

This schema is useful for generating a global view of the system. It lists all the theories in the VAMP, grouped by their context. A unique string for addressing each theory can be constructed using the convention: *context_id@theory_id*.

### 4.2.2. The Imported Theories Schema and JGraph

Table 4.2. The Imported Theories Schema

| **XML Tag Name** | **Level** | **Attributes** | **Comments** |
|---|---|---|---|
| <gxl> | 1 | - | Root node; defines the namespace |
| \| – <graph> | 2 | id | Each file contains one graph for an entire context |
| \| – – <node> | 3 | id | One for each imported theory or context |
| \| – – – <attr> | 4 | name="Label" | Identify this node using a string |
| \| – – – – <string> | 5 | - | *context@theory* for theories; or just *context* |
| \| – – – <attr> | 4 | name="Bounds" | The coordinates of this node |
| \| – – – – <tup> | 5 | - | Group the attributes for left, right, width and height |
| \| – – – – – <int> | 6 | - | |
| \| – – – <attr> | 4 | name="ExtraLabels" | *optional*; (node is a context) list the theories in this context that are imported. |
| \| – – – – <tup> | 5 | - | |
| \| – – – – – <string> | 6 | - | |
| \| – – – – – – <from> | 7 | - | |
| \| – – – – – – <to> | 7 | - | |

This schema is basically the directed acyclic graph (DAG) layout encoded using the GXL standard [GXL]. Each graph represents an entire context and describes the relationships between the theories in the context. Relationships to theories in another context are grouped under one node labeled by the target context and listed using the *ExtraLabels* attribute. We limit our DAG layout to individual contexts because the graph showing the relationships between all theories in the VAMP is simply too large and has too many connections to be useful to a user.

Each XML file is generated from a graph created using the JGraph Java libraries. JGraph is designed as a "Swing compliant implementation of a graph component"

[Al03]. In other words, it combines the architecture found in the standard Java Swing user interface library with concepts from graph theory such as vertices, edges and algorithms for layout and traversal. We use a layout algorithm that is optimized for hierarchical graphs.

Generating and encoding the graphs in the data extraction phase saves us from doing this at run-time in the visualization phase. This optimization simplifies the corresponding visualization components and makes the interface more responsive.

### 4.2.3. The Theory Components Schema

This schema provides details about all the elements in a PVS theory including the code used to specify these elements and the class they belong to. It supports the Local view in Figure 3.6 by listing the theories imported by this theory, the theories that used this theory and the components in the theory. It also supports the Content view by providing a syntax-colored representation of each element's specification.

The syntax coloring feature is initiated by several XML tags which markup each token in the content. The actual formatting details and colors can be provided using a stylesheet in the visualization phase. This gives designers in the visualization phase more flexibility to choose and change the syntax coloring format.

Table 4.3. The Theory Components Schema

| **XML Tag Name** | **Level** | **Attributes** | **Comments** |
|---|---|---|---|
| <theory> | 1 | - | Root node; |
| \| – <header> | 2 | - | Specify some basic properties of the theory. |
| \| – – <context> | 3 | - | Its context |
| \| – – <filename> | 3 | - | The PVS specification file |
| \| – – <theoryformals> | 3 | - | *optional*; The theory's parameters. |
| \| – – – <theoryformal> | 4 | id | |
| \| – – – – <string> | 5 | - | |
| \| – – – – <importing> | 5 | - | |
| \| – <imports> | 2 | - | *optional*; List all the theories imported by this theory. Each imported theory is contained in an <importing> object and expressed using its context as well as its theory name. |
| \| – – <importing> | 3 | id | |
| \| – – – <string> | 4 | - | |
| \| – – – <context> | 4 | - | |
| \| – – – <theoryname> | 4 | - | |
| \| – <usedbylist> | 2 | - | *optional*; List all the theories that use this theory. Each *usedby* theory is contained in a <usedby> object and expressed using its context as well as its theory name. |
| \| – – <usedby> | 3 | id | |
| \| – – – <context> | 4 | - | |
| \| – – – <theoryname> | 4 | - | |
| \| – <components> | 2 | id, label | List all the theory elements, grouped by the section they occur in (the Exporting Part, Assuming part or Theory Body). <string> specifies a label for each element, while <type> is used to classify the element (as Library, Constant, Lemma etc.). <content> is marked up with the following syntax tags: |
| \| – – <component> | 3 | id | |
| \| – – – <string> | 4 | - | |
| \| – – – <type> | 4 | - | |
| \| – – – <content> | 4 | - | |
| \| – – – – <comment> | 5 | - | Every thing following a '%' token |
| \| – – – – <group> | 5 | - | '(' , ')', '[', '(#' and other parentheses |
| \| – – – – <oper> | 5 | - | +, –, /, & and other operators |
| \| – – – – <keyword1> | 5 | - | Keywords for major groups like Library, Theory etc. |
| \| – – – – <keyword2> | 5 | - | Other keywords: BEGIN, TRUE, ALL etc. |
| \| – – – – <idtoken> | 5 | - | Corresponds to the *Id* token in the PVS grammar |
| \| – – – – <number> | 5 | - | Corresponds to the *Number* token in the grammar |

### 4.2.4. The Schematic View Schema

This schema is intended to support the schematic view. To do this, each component must be broken into its constituent parts, arranged in a graph and encoded using a GXL format similar to that of Table 4.2. The visualization phase can then replace the nodes in this graph with icons that make it look more like a schematic. This includes icons for AND gates, flip-flops and multiplexers.

### 4.2.5. PVS Proofs Scripts

A PVS proof file is associated with every specification file. It contains proof scripts for each formula in the theory that has been proved. The scripts are written using a Lisp format. It is quite simple to create a grammar to parse proof scripts – one is shown in Appendix B. This can be used to extract the commands used in proofs and structure them for display in a graphical format. But we do not use this at this point in the development of the VAMP explorer, choosing instead to focus on the presentation of the theories.

### 4.2.6. The Directory Structure

Figure 4.3 shows the final directory structure used to order the XML file created in this phase. When multiple files are needed to support a view, the files are arranged in folders and files whose names mirror the context and theory names respectively in PVS. This allows us to *address* an xml file based on the context or theory it is supposed to represent. In the visualization phase, we will assume this addressing scheme when creating hyperlinks between different components and theories.

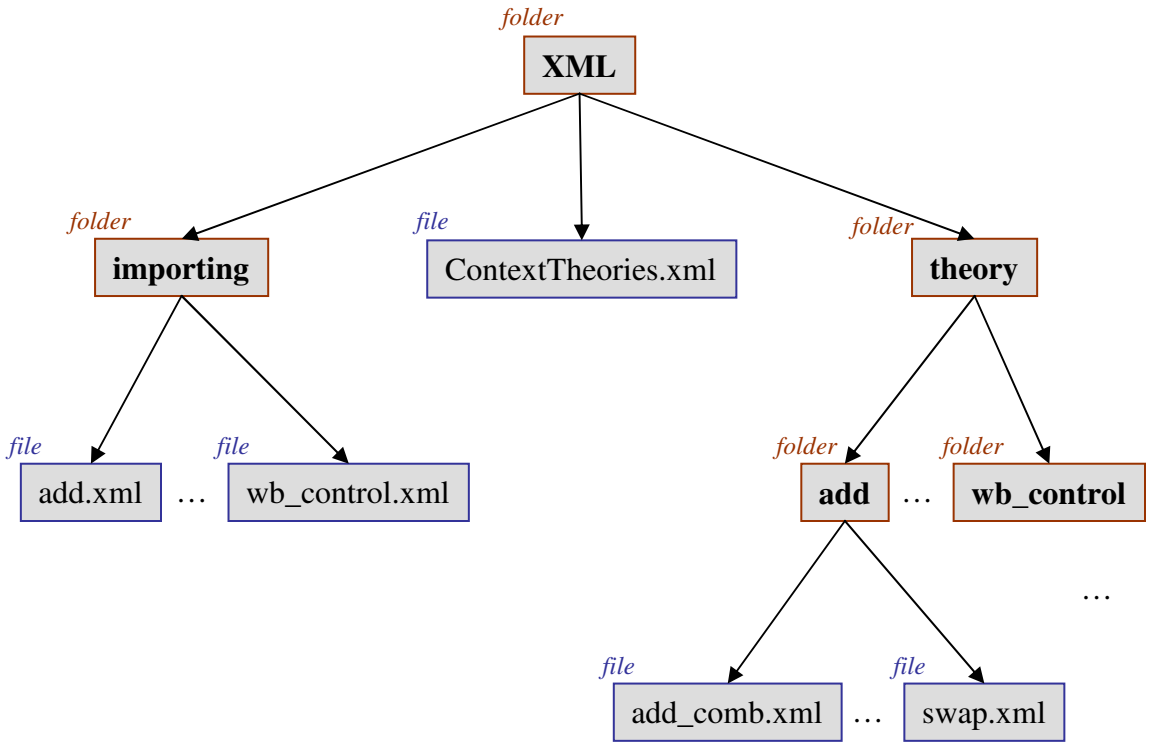



Figure 4.3. Planning the Directory Structure for the XML Files

## 4.3. Java Implementation of Data Extraction Algorithms

The data extraction effort is performed by a suite of Java classes build around the framework of the GOLD parser engine. This suite elegantly prepares all the XML files needed to support the visualization phase using the existing PVS files without modifying them. It is designed to be general and applicable to other projects involving PVS files arranged in contexts.

### 4.3.1. Class Diagram

Figure 4.4 shows a high level class diagram outlining the relationships between the data extraction classes. The start point for the system is DataExtraction which coordinates

the data extraction process. The workhorse classes are ContextTheories, ImportingParser and ProcessTheories which prepare different schemas. They depend on a number of interfaces and factory classes displayed at the top of the diagram. Other dependencies include the Gold Engine package which contains classes that use the LALR table created by the GOLD parser to parse PVS specification files.

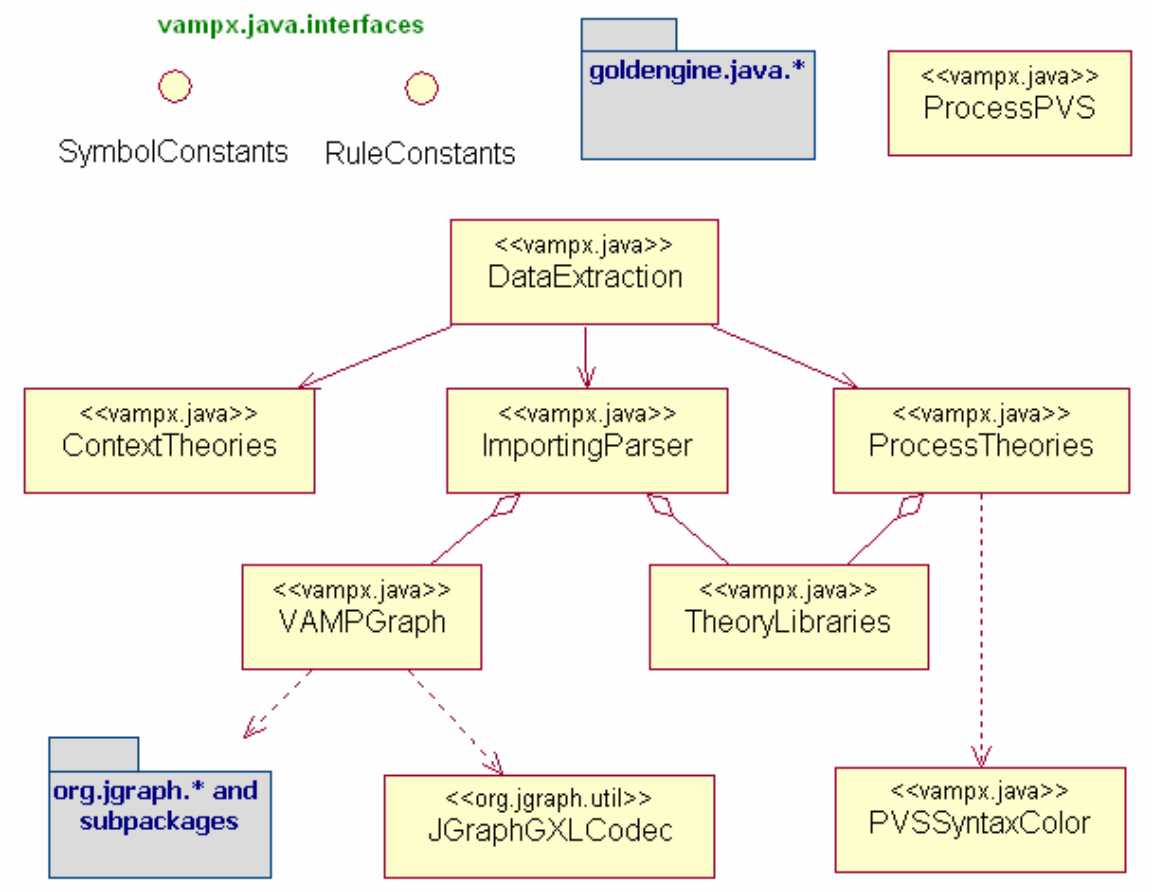



Figure 4.4. Class Diagram for Data Extraction Phase

### 4.3.2. The Data Extraction Algorithm

Given the classes defined above, we carry out the data extraction using a few simple steps:

1. Specify the root directory of the PVS project (in this case the VAMP files).
2. Run *ContextTheories* to create a list of all theories in this context.
3. Run *ImportingParser* to extract the importing relationships in each context and lay them out in a graph. This step simultaneously creates a mapping from each theory to every theory that uses it.
4. Run *ProcessTheories* to prepare a list of all components in each theory. Each theory also captures a list of the theories that use it from the mapping in step 3 and a list of the theories it imports.

In the next chapter, we describe how this data is used to visualize the VAMP.

Chapter 5

# THE VAMP EXPLORER USER INTERFACE

## 5.1. Planning the Interface

As we set out to create a user interface for the VAMP theories, we must keep the needs of the user in focus. Our implementation is intended as a prototype to provide a framework for visualizing PVS theories and may be limited by time and development constraints. But it should still facilitate communication between theory writers and readers by exposing the big picture and providing access to related components, theories and other resources.

### 5.1.1. Why Flash?

Our interface is developed using Macromedia Flash MX 2004 Professional edition. Flash was introduced to provide animations and dynamic content over the World Wide Web. It has evolved to be one of the most effective ways to deliver applications and interfaces over the web. Our reasons for choosing Flash include:

- It is effective as a rapid prototyping and development environment when experimenting with visual interfaces.
- It is readily available over the web, with over 90% of web users equipped to run Flash applications [Mac05].

- It delivers a consistent and lightweight experience across diverse platforms and browsers.
- The Flash IDE comes with customizable modern user interface (UI) components such as menus, trees, scroll panes and combo-boxes. It is also supported by a vibrant community of professionals and enthusiasts who regularly increase the variety of components available to interface designers. (Here, the word 'components' refers to widgets used to build the user interface. This should not be confused with our earlier use of the word to describe the parts of the theory.)
- It provides strong support for XML, web services and other standards.
- The latest versions of Flash provide an object-oriented language called Actionscript 2.0 that is similar to Java and that helps make Flash applications more modular, structured and extendible.

The main disadvantage is that the Flash IDE is not freely available to developers that may want to modify our implementation, but it is relatively inexpensive for educational users. In addition, researchers that which to use the framework on projects other than the VAMP do not need to modify any of the Flash files since the displayed content is dynamically loaded from data files.

### 5.1.2. Design Paradigm

We aim to produce a modular and loosely coupled implementation. This paradigm leads us to design our system as a collection of independent Flash applications (called movies) representing each of the different views and connected through a main class.

Figure 5.1 shows a class diagram showing the relationships between Actionscript classes and associated movies.

Flash provides and *listener/broadcaster* model as the preferred means of communication between components and movies. We build upon this functionality by using *events* broadcast each time certain actions occur in one movie to notify other relevant movies of these actions.

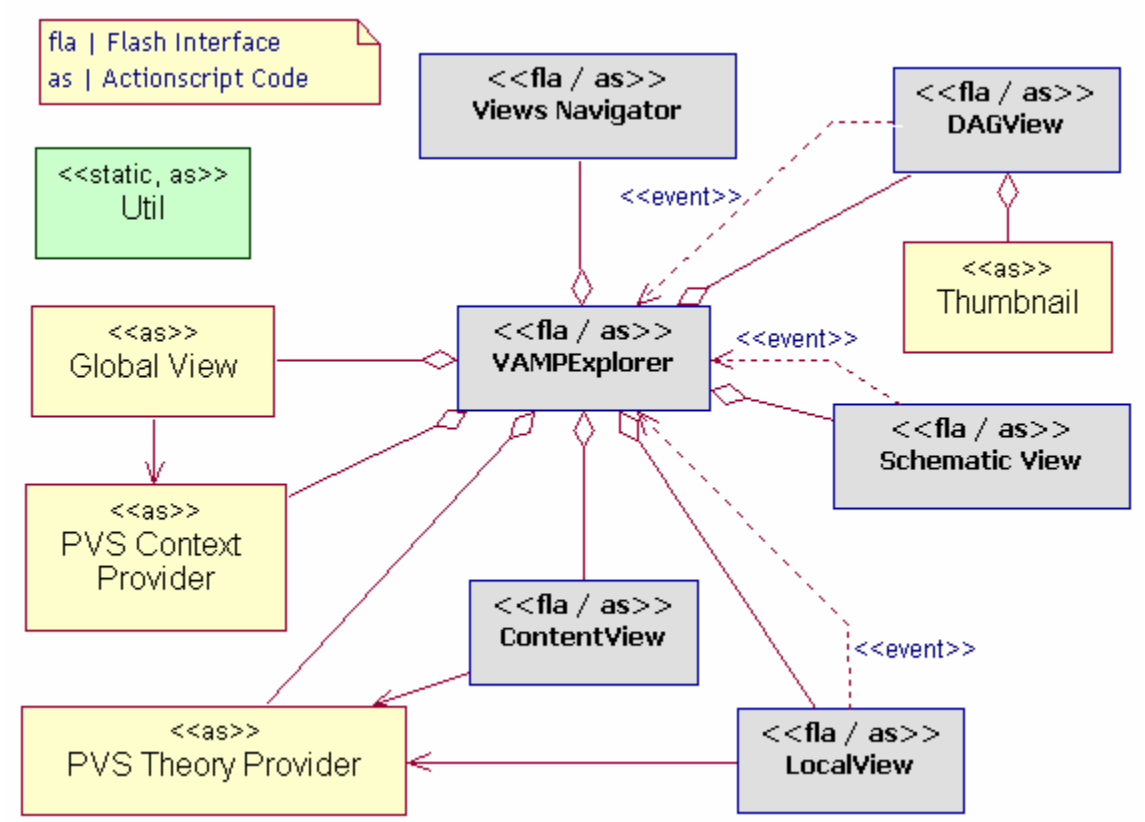



Figure 5.1. Class Diagram for Flash Actionscript Classes

## 5.2. Providing Global Access to all Theories

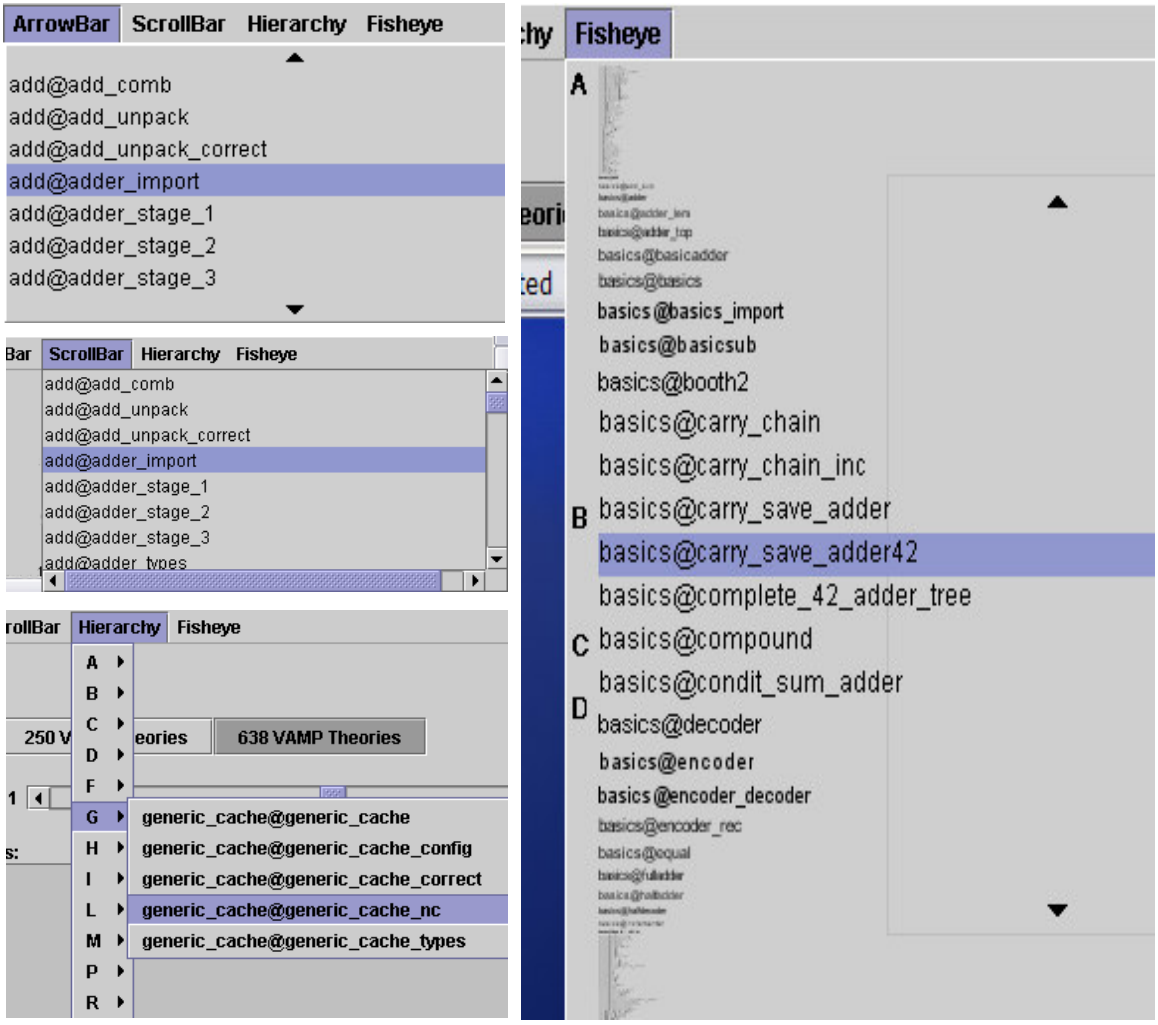
The *GlobalView* class is not associated with any independent movies. It manipulates components defined in the main VAMPExplorer application. It gives a user quick access to any theory in the system using a menu system. Menus promote *recognition* over *recall* – i.e. the user does not have to remember the name of the theory. However, given the large number of theories to be considered, we need to consider what the most effective menu-based approach is.

### 5.2.1. Visualizing Long Lists with Menus

Reference [Be00] describes a number of menu-based approaches and proposes a *Fish-eye* menu as an effective way to navigate large amounts of information. Figure 5.2 illustrates the different menus. The *Arrowbar* and *ScrollBar* based menus allow a user to *explore* the information by scrolling up and down but are not very effective for user's that know what they are looking for and want instant access to it. The *Hierarchy* menu breaks up the content into categories by first name and lists the elements in each category. Like a dictionary, it is quite effective for someone wanting quick access to a theory but is quite clumsy for exploring.

The Fish-eye menu combines the strengths of the above menus by making content close to the mouse pointer large and gradually reducing the size of menu items as they move away from the pointer. Users can explore the list by moving the mouse slowly over the menu, or they can quickly access a particular item by moving the mouse into the vicinity of the item and then freezing the fish-eye lens (by moving over the right portion

of the menu) and accessing the desired item. A demo of the fish-eye menu is available at [Be00].




Source: http://www.cs.umd.edu/hcil/fisheyemenu/

Figure 5.2. Menus for Visualizing Long Lists

## 5.3. Creating a Directed Acyclic Graph View

The directed acyclic graph (DAG) view is the most intuitive for the casual user. It uses nodes to represent each of the theories and edges to indicate when one theory imports another. The nodes are arranged in layers such that the node at the lower part of an edge (the child) is always imported by the node at the upper part of that edge (the parent).

### 5.3.1. DAG view Challenges

Two problems arise with this view of the theories. One is that a large number of nodes will not easily fit in the viewing area of the user's screen. We alleviated this problem somewhat in the data extraction phase by limiting the graph to a single context, but that still produces a large number of nodes for some contexts. So in the DAG view, we provide a thumbnail window that allows the user to see the big picture and navigate over all the nodes. We also provide a zooming utility that allows the user to increase or decrease the range of visibility of the nodes. Finally, we place the entire view in a scrolling window component that allows users to navigate to the hidden portions of the graph.

Another problem with the DAG view that is quite pronounced in many of the VAMP contexts is the large number of interconnections that may occur between the nodes. When this happens it can be quite difficult to determine which nodes are related. We can alleviate this problem by making the entire interface semitransparent so that all edges are visible and then highlighting all the edges that are connected to the selected node.

To further reduce the number of nodes in each graph, we group all the theories that are imported from a different context into one node labeled by the context name. The user can activate a pop-up menu in these context nodes to see details about which theories in this context are imported. Selecting the context node causes a new DAG graph for that context to be loaded. In this way a user can navigate through the VAMP.

Figure 5.3 illustrates some of the features in the DAG view.

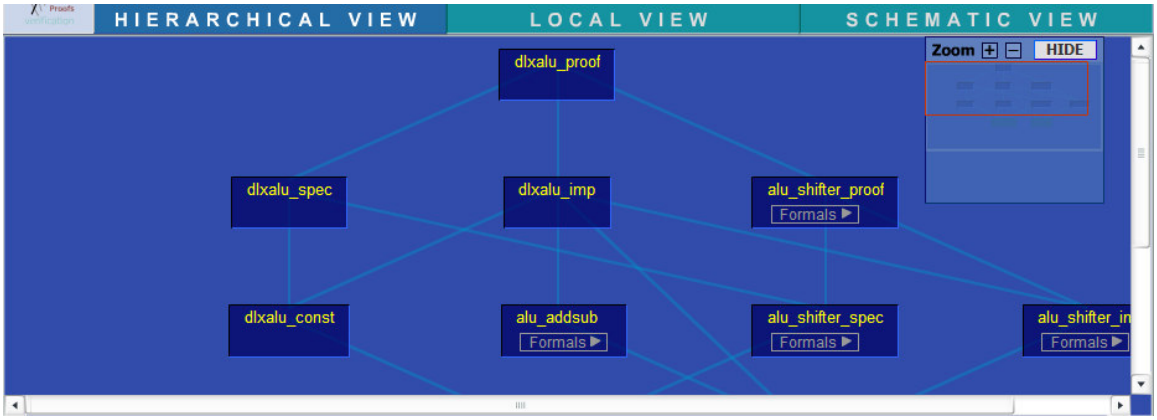



Figure 5.3. The Directed Acyclic Graph View

### 5.3.2. DAG view Actions

The DAGView class manages its own connections to the XML files that layout the graphs. Each time a context node is clicked, the XML data connector connects over the web to the XML file representing that context and loads it. When it has finished loading, it broadcasts an event which is received by the class and used to redraw the graph.

Whenever any node is selected DAGView broadcasts a ‘clicked’ event which can be received by the main VAMPExplorer class and used to drive activities in the other views. For example, the content view can load the code contained in the theory that was clicked.

## 5.4. Creating a Local View

The local view presents the VAMP from the perspective of a single theory. It shows all the subcomponents in this theory as well as its immediate child (importing) and parent (used by) relationships. A user can navigate through this section of the VAMP hierarchy by selecting the related theories. This results in a more narrow focus in the hierarchy and is most useful when a user wants to focus on a small subset of the tree.

### 5.4.1. Choosing Local View Components

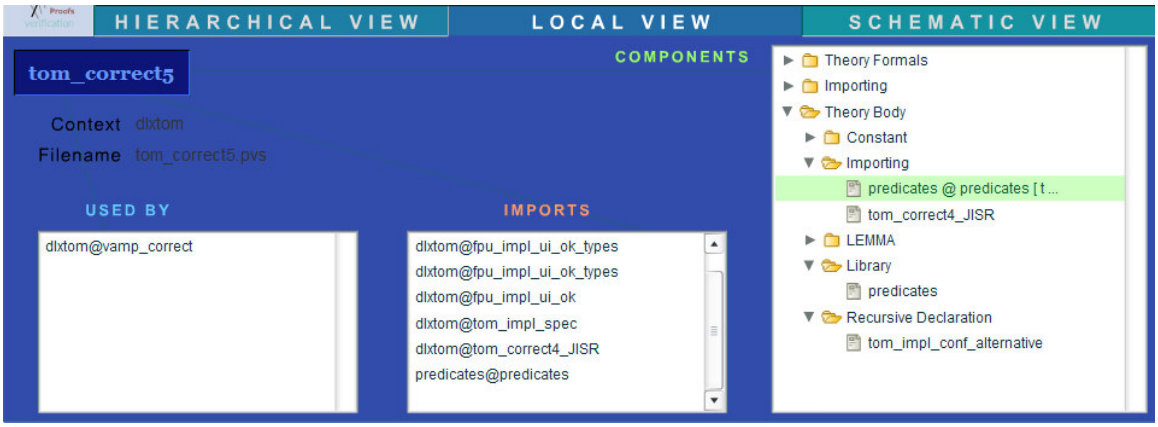



Figure 5.4. The Local View for the *tom_correct5* Theory

Figure 5.4 shows a screen shot of the local view for the theory *tom_correct5*. Simple *list* components are used to represent the 'used by' and 'importing' fields because these are just one dimensional fields and do not contain a large number of values. Each theory is presented using the full url: context@theory. This way, no additional information about the location of the corresponding theory is needed.

The ‘components’ field is more complex because each theory element is classified based on its type and which part of the theory it occurs in. To bring out this hierarchy, we use the Flash Tree UI Component. This component will be familiar to many users as the interface used to access the file-system on most GUI based operating systems. Users can show or hide subsets of the hierarchy as they seek instant access to a desired element. As the tree grows, the component transforms into a list-like component with scrollbars to provide access to hidden parts of the tree.

### 5.3.2. Local view Actions

The Local view broadcasts two kinds of events to the other views. When a theory is selected in the either the ‘usedby’ or ‘importing’ list box, a ‘clicked’ event is broadcast which causes most of the other views to update themselves since a new theory is launched. This can be seen as a global event. On the other hand, any action on the tree will result in a more local ‘treeChange’ event which only updates the content view. Both of these events are handled by the main VAMPExplorer class which is then responsible for initiating changes in the appropriate views.

The XML files that provide content for the local view are also used by the content view. For this reason, we pass the responsibility for connecting to the XML files to a “neutral” class: *PVSTheoryProvider*. An instance of this class is contained in the main VAMPExplorer class and fed to both the local and content view to provide the necessary content.

## 5.5. Creating a Schematic View

The purpose of the schematic view is to provide more detail about each of the theory elements by presenting them in a schematic. This is particularly applicable to the VAMP project since many of the elements represent hardware circuits and gates.

Interestingly, the implementation for this view is very similar to the DAG view. Most of the effort is shifted to the data extraction phase which must identify the parts of each component and 'graph' them into a layout that represents a schematic. The visualization phase, like the DAG view, simply publishes this layout and provides actions each time a node is selected. The only difference is that in the schematic view, each node should be represented by an icon that reflects a traditional understanding of the type of circuit contained in that node. This includes icons representing AND-gates, multiplexers and other circuits. Figure 5.5 lists some of these icons and the circuits they represent.

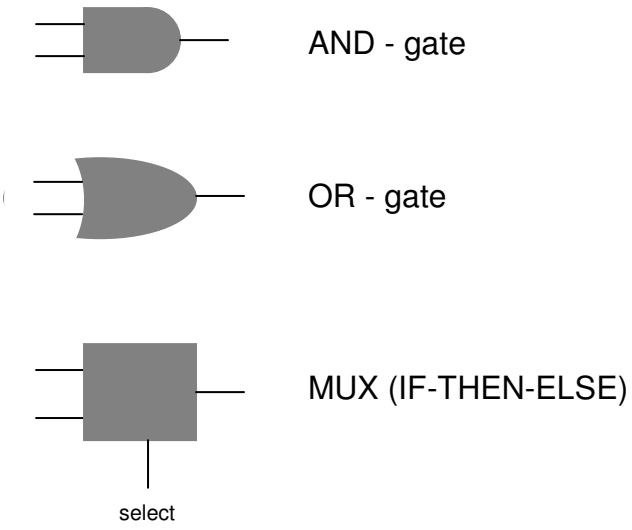



Figure 5.5. Representing Components in the Schematic View

## 5.6. Creating a Content view

The content view is in many ways the center of attention for most users because it contains the specification they will use to understand the theory. The content view provides features that facilitate the user's efforts to do this.

### 5.6.1. The Split Screen

Figure 5.7 shows a screenshot of the content view in split screen mode. This mode allows users to perform side-by-side comparison of different parts of a theory or different theories altogether. Users can freely switch between this mode and the single screen mode shown in Figure 5.6.

To facilitate this feature, we introduce the concept of an 'active pane'. The active pane is simply the screen that is currently selected. This is shown visually by using bolder colors to identify the pane. Internally, a variable keeps track of which pane is active. When an event in one of the other views triggers a change in the content view, that change always takes place in the active pane. Additionally, when a user switches into single screen mode, the contents of the active pane are preserved and displayed. (The contents of the inactive pane are also preserved but are hidden.) Finally, the control panel is updated each time the user selects a new pane to reflect the theory in that pane.

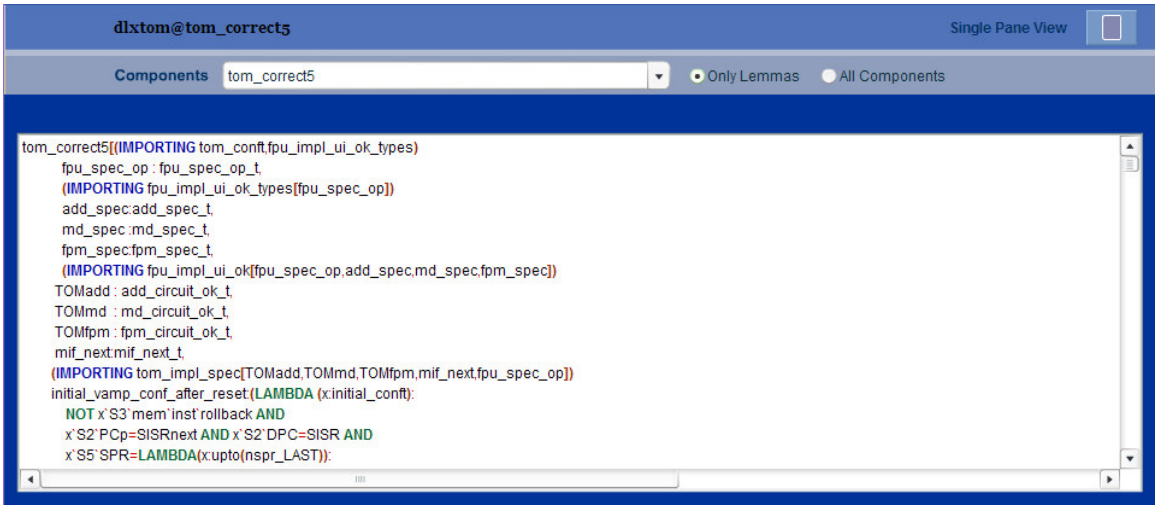



Figure 5.6. The Content View: Single Screen Mode

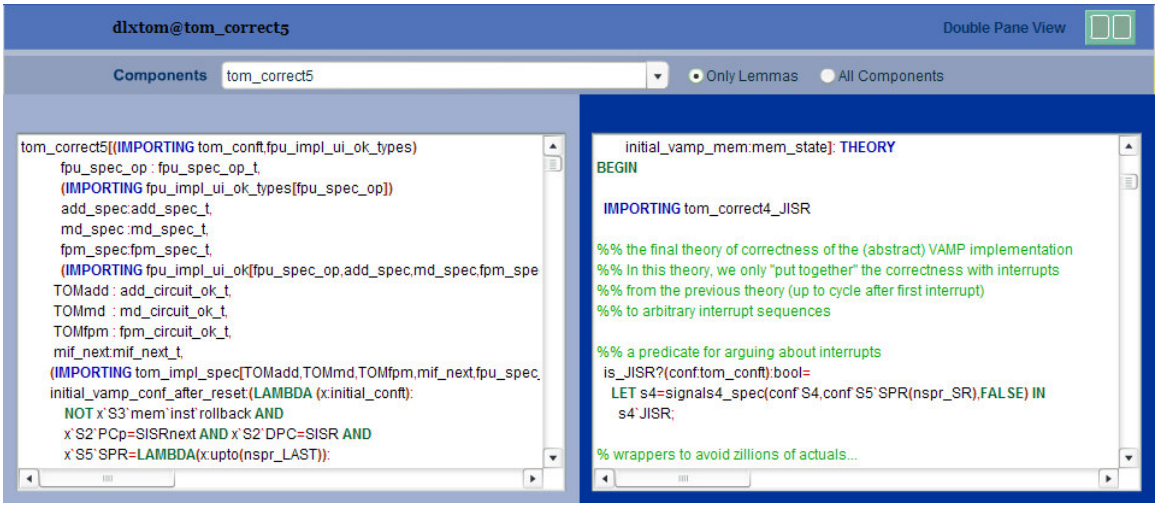



Figure 5.7. The Content View: Split Screen Mode

One important implication of the split screen view is that it is possible for the inactive pane to contain content from an ‘old’ theory, that is, a different theory from the one loaded into the main VAMPExplorer class and all the other views. Therefore, facilities have to be built into the content view to support local changes it may want to make

through its control panel. For example, if the inactive pane is activated and a new component is selected from the control panel, its code must be derived from the old theory, not the one currently loaded into VAMPExplorer.

### 5.6.2. The Control Panel

The control panel provides some autonomy to the content view to allow the user to explore a particular theory. It features a Flash drop-down combo box UI component from which the user chooses an element to view. The list can be filtered to contain only elements of the same type as the currently displayed element using the provided radio boxes.

### 5.6.3. The History Buttons

The concept of a history will be familiar to most web users because this is a standard feature on all browsers. Keeping track of the history of 'places' the user has visited is complicated by the presence of two panes. Each pane keeps track of its own history and the history buttons always refer to the active pane.

The content view acts as a slave to the other views. Actions in these views can lead to changes in the content displayed but the content view cannot change the global state of the system or launch new theories in any of the other views. We create this arrangement so that users can freely move back and forth in the history without needing to launch previously viewed theories. Again we have the problem described in section 5.6.1 of old theories appearing in the content pane.

### 5.6.4. Other Content View Features

Even though the content view cannot effect changes in the other views, it does broadcast events in response to actions on the combo box, the radio boxes and the history and pane control buttons. These events are handled locally (though occasionally the history feature may initiate a request to the PVSTheoryProvider to reload a copy of an old theory).

The content view uses stylesheets to perform syntax coloring on the code according to the divisions first introduced in the data extraction phase (Table 4.3). Figure 5.8 gives an example of a stylesheet that assigns formatting properties to each of the markup tags that can appear in the content field. The stylesheets are attached to the Flash TextArea UI component that takes up most of the content view and displays the PVS specification.

```
content  { color : purple; }
comment  { color : green;  }
group    { color : brown;  }
oper     { color : red;    }
keyword1 { color : blue;   }
keyword2 { color : green;  }
idtoken  { color : black;  }
number   { color : black; font-weight:bold; }
string   { color : orange; }
```

Figure 5.8. A Stylesheet to Facilitate Syntax-Coloring

## 5.7. Bringing It All Together

Some of the roles of the VAMPExplorer class have already been mentioned in previous sections. This class is associated with a movie that loads the other independent movies and it responds to events these movies generate. It ensures all the views are

updated to contain the global current theory (including the active pane in the content view). It also synchronizes changes within the views so that they do not conflict. For example, it is problematic to have multiple connections to XML files from different movies loaded into the main movie. So VAMPExplorer must ensure these connections occur sequentially. It does this by listening for events broadcast each time a data source is completely loaded.

Figure 5.9 gives an overview of the VAMPExplorer bringing together all the movies. Since the content view is likely to be constantly relevant to the user, it is always visible in the lower half of the window. The other views are displayed in the upper half of the window when the respective tabs are selected. The colors used in all the views are consistent so that the entire interface appears to the user as one continuous entity.

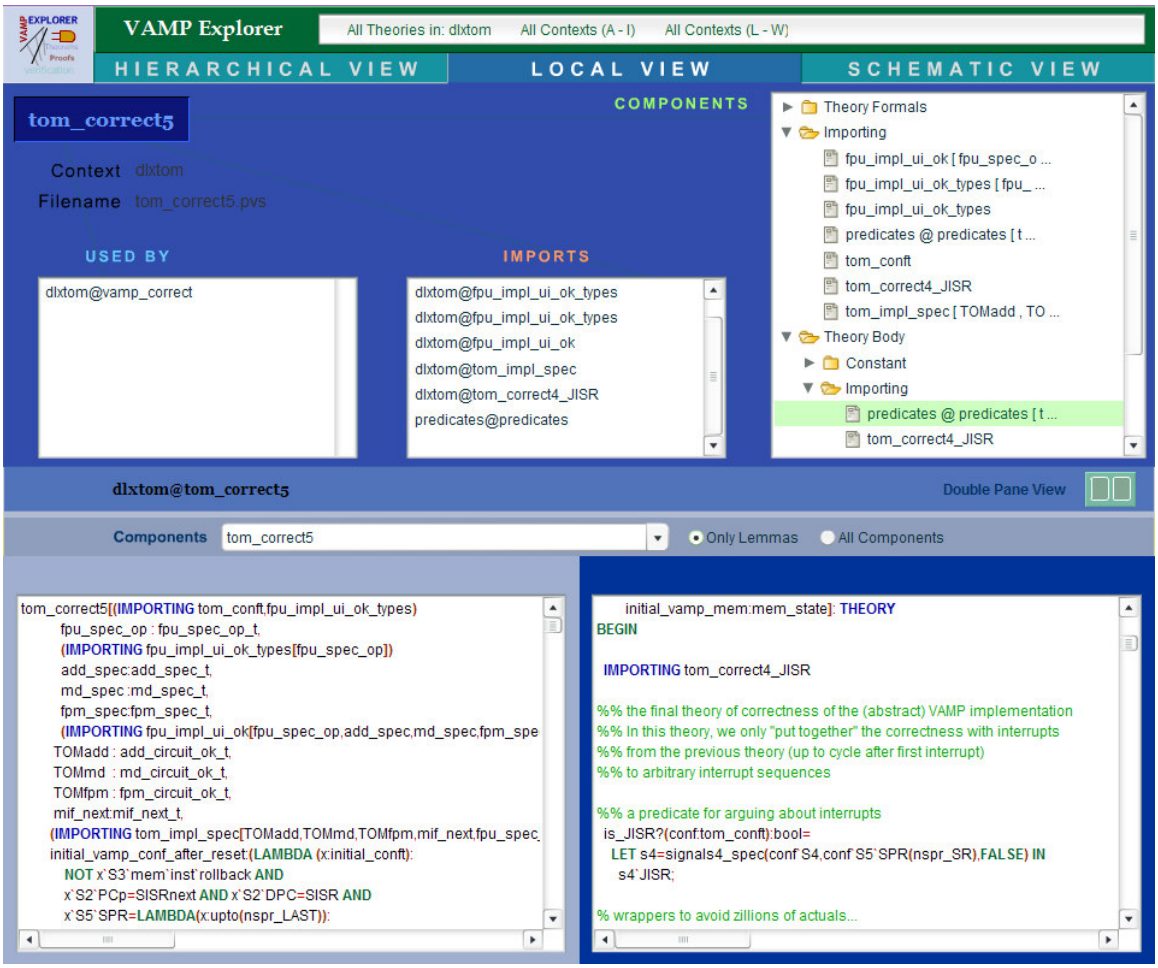



Figure 5.9. Overview of the VAMP Explorer

# Chapter 6

# DEPLOYING THE VAMP EXPLORER

## 6.1. Full System Deployment

The VAMP Explorer was designed to be general so it can be applied to other PVS projects. It makes some assumptions about the target project but does not rely on VAMP specific constructs. The data extraction phase assumes the target project is arranged in directories representing contexts and containing PVS specification files. The system allows multiple theories in a specification file.

The data extraction modules are implemented in a platform independent way using Java. All the user needs to do is supply a root directory where the PVS files are and a target directory where the XML files will be stored.

### 6.1.1. Web Deployment

Once the XML directory has been generated, it should be placed in the folder that contains all the other web specific files. The include SWF (Shockwave Format) files generated by Flash, HTML files, Cascading Style Sheet files and configuration files. The installer does not need to recreate the SWF files, nor does he or she need access to the Flash IDE. This is because the content displayed in the movies is dynamically loaded from the XML files. Figure 6.1 shows the final directory structure for this web folder.

The entire folder can be transferred to a web server to make the project accessible over the web.

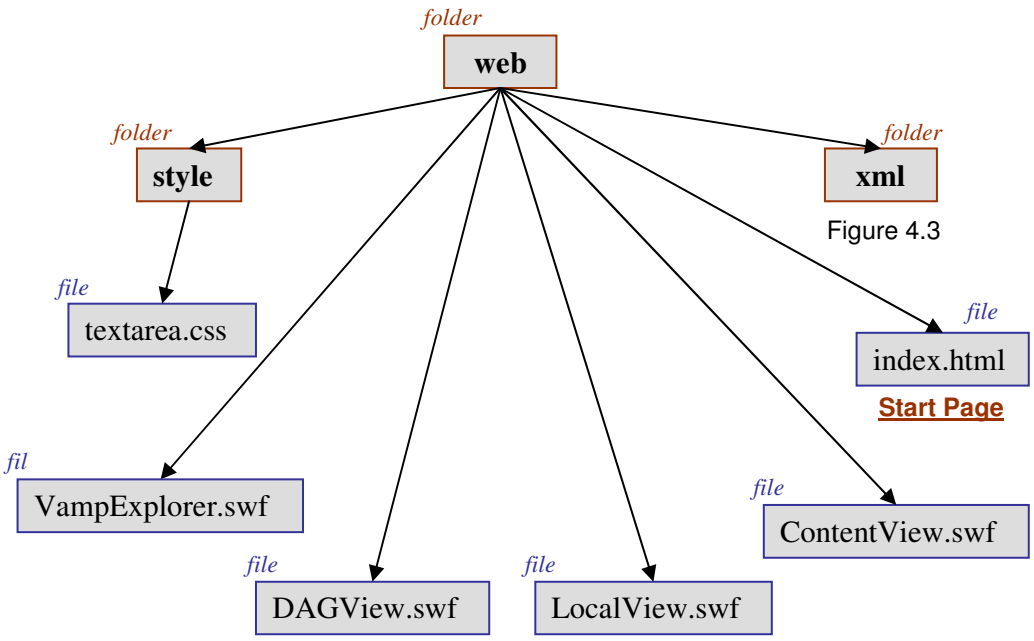



Figure 6.1. Directory Structure for the Web Folder

# Chapter 7

# CONCLUSION AND FUTURE DIRECTIONS

VAMP Explorer is a resource for researchers looking for ways to visualize or communicate about theories created using PVS. This project has demonstrated some visualization approaches and provided a framework for applying them to a wide variety of projects. It has also revealed some potential areas of growth for the system.

One important aspect that was not explored is a documentation system. The VAMP theories do not contain comments structured according to any particular set of rules. The in-code commenting style exemplified by Javadoc [Sun05] is the most likely candidate for a documentation system which can be integrated into the VAMP Explorer interface.

This project also does not explore different methods for visualizing proof trees. The PVS system already visualizes proofs using a tree diagram to display the hierarchy of commands with some arranged sequentially and others on the same level.

The user's access to theories could be improved by making it possible to run queries on the system. The queries can be simple searches for theories by name or more complicated queries over the XML files that form the basis for the visualization. In addition, we could provide hyperlinks within the theory code that link relevant keywords to their corresponding theories.

To make the VAMP Explorer prototype a more widely used ‘PVS Explorer’, it will need to be refined and tested with other projects. This action may expose more details and nuances that do not occur in the VAMP but that may apply in other projects and scenarios.

## APPENDIX A. Modified BNF Form of the PVS Specification Language

```
! ----------------------------------------
! -- M o d i f i c a t i o n  of pvs.bnf
! ----------------------------------------
!    Express pvs.bnf in BNF form (not extended)
! ----------------------------------------

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! First some GOLD Parser settings
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

"Name"     = 'PVS'
"Author"   = 'Nathaniel Ayewah'
"Version"  = 'PVS 2.3'
"About"    = 'The PVS 2.3 Grammer in BNF form for use with the Gold Parser'

"Start Symbol" = <Specification>
Comment Line = '%'

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

<Specification> ::= <Theory> | <Datatype>
            |  <Theory> <Specification>
            |  <Datatype> <Specification>

! SHIFT-REDUCE Correction !
<Theory>  ::=  <IdOp>  <TheoryFormals>  ':'  'THEORY'  <TheoryOption2>  'BEGIN'  <TheoryOption3>
<TheoryOption4> 'END' Id
       |  <IdOp> ':' 'THEORY' <TheoryOption2> 'BEGIN' <TheoryOption3> <TheoryOption4> 'END' Id
<TheoryOption2> ::= <Exporting> |
<TheoryOption3> ::= <AssumingPart> |
<TheoryOption4> ::= <TheoryPart> |

<TheoryFormals> ::= '[' <TheoryFormalList> ']'
<TheoryFormalList> ::= <TheoryFormal> ',' <TheoryFormalList> | <TheoryFormal>

<TheoryFormal> ::= '(' <Importing> ')' <TheoryFormalDecl> | <TheoryFormalDecl>

<TheoryFormalDecl> ::= <TheoryFormalType> | <TheoryFormalConst>

! CORRECTION !
<TheoryFormalType> ::= <IdOps> ':' <TypeDeclGroup2> 'FROM' <TypeExpr>
               |  <IdOps> ':' <TypeDeclGroup2>
```

```
<TheoryFormalConst> ::= <IdOps> ':' <TypeExpr>

<Exporting> ::= 'EXPORTING' <ExportingNames> 'WITH' <ExportingTheories>
        |  'EXPORTING' <ExportingNames>

<ExportingNames> ::= 'ALL' 'BUT' <ExportingNameList>
            |  'ALL'
            |  <ExportingNameList>
<ExportingNameList> ::= <ExportingName> ',' <ExportingNameList> | <ExportingName>

<ExportingName> ::= <IdOp> <ExportingNameOptions>
<ExportingNameOptions> ::= ':' <TypeExpr> | ':' 'TYPE'
                |  ':' 'FORMULA'  |

<ExportingTheories> ::= 'ALL' | 'CLOSURE' | <TheoryNames>

<Importing> ::= 'IMPORTING' <TheoryNames>

<AssumingPart> ::= 'ASSUMING' <AssumingElementList> 'ENDASSUMING'
<AssumingElementList> ::= <AssumingElement> <Semicolon> <AssumingElementList>
              |  <AssumingElement> <Semicolon>

<AssumingElement> ::= <Importing>
                    |  <Assumption>
                    |  <TheoryDecl>

<TheoryPart> ::= <TheoryElementList>
<TheoryElementList> ::= <TheoryElement> <Semicolon> <TheoryElementList>
              |  <TheoryElement> <Semicolon>
<Semicolon> ::= ';' |

<TheoryElement> ::= <Importing> | <TheoryDecl>

<TheoryDecl> ::= <LibDecl>
        |  <TheoryAbbrDecl>
            |  <TypeDecl>
            |  <VarDecl>
            |  <ConstDecl>
        |  <RecursiveDecl>
            |  <InductiveDecl>
            |  <FormulaDecl>
            |  <Judgement>
            |  <Conversion>
            |  <InlineDatatype>

! CORRECTED !
<LibDecl> ::= <IdOps> ':' 'LIBRARY' '=' String
      |  <IdOps> ':' 'LIBRARY' String

! CORRECTED !
<TheoryAbbrDecl> ::= <IdOps> ':' 'THEORY' '=' <TheoryName>

! CORRECTED !
```

```
<TypeDecl> ::= <IdOps> ':' 'TYPE' <TypeDeclGroup3>
        |  <IdOps> ':' 'NONEMPTY_TYPE' <TypeDeclGroup3>
        |  <IdOps> ':' 'TYPE+' <TypeDeclGroup3>
        |  <IdOp> <Bindings> ':' 'TYPE' <TypeDeclGroup3>
        |  <IdOp> <Bindings> ':' 'NONEMPTY_TYPE' <TypeDeclGroup3>
        |  <IdOp> <Bindings> ':' 'TYPE+' <TypeDeclGroup3>
        |  <IdOp> ':' 'TYPE' <TypeDeclGroup3>
        |  <IdOp> ':' 'NONEMPTY_TYPE' <TypeDeclGroup3>
        |  <IdOp> ':' 'TYPE+' <TypeDeclGroup3>
<TypeDeclGroup2> ::= 'TYPE' | 'NONEMPTY_TYPE' | 'TYPE+'
<TypeDeclGroup3> ::= <TypeDeclGroup4> <TypeExpr> 'CONTAINING' <Expr>
            |  <TypeDeclGroup4> <TypeExpr>
            |
<TypeDeclGroup4> ::= '='|'FROM'

<VarDecl> ::= <IdOps> ':' 'VAR' <TypeExpr>

<ConstDecl> ::= <IdOps> ':' <TypeExpr> '=' <Expr>
        |  <IdOp> <Bindings> ':' <TypeExpr> '=' <Expr>
        |  <IdOp> <BindingsPlus> ':' <TypeExpr> '=' <Expr>
        |  <IdOp> ':' <TypeExpr> '=' <Expr>
        |  <IdOps> ':' <TypeExpr>
        |  <IdOp> <Bindings> ':' <TypeExpr>
        |  <IdOp> <BindingsPlus> ':' <TypeExpr>
        |  <IdOp> ':' <TypeExpr>

<RecursiveDecl> ::= <IdOps> ':' <RecursiveDeclEnd> <MeasureDeclEnd>
            |  <IdOp> <Bindings> ':' <RecursiveDeclEnd> <MeasureDeclEnd>
            |  <IdOp> <BindingsPlus> ':' <RecursiveDeclEnd> <MeasureDeclEnd>
            |  <IdOp> ':' <RecursiveDeclEnd> <MeasureDeclEnd>
<RecursiveDeclEnd> ::= 'RECURSIVE' <TypeExpr> '=' <Expr>
<MeasureDeclEnd> ::= 'MEASURE' <Expr> 'BY' <Expr> | 'MEASURE' <Expr>

<InductiveDecl> ::= <IdOps> ':' <InductiveDeclEnd>
            |  <IdOp> <Bindings> ':' <InductiveDeclEnd>
            |  <IdOp> <BindingsPlus> ':' <InductiveDeclEnd>
            |  <IdOp> ':' <InductiveDeclEnd>
<InductiveDeclEnd> ::= 'INDUCTIVE' <TypeExpr> '=' <Expr>

<BindingsPlus> ::= <Bindings> <BindingsPlus> | <Bindings>

! CORRECTED !
<Assumption> ::= <IdOps> ':' 'ASSUMPTION' <Expr>

! CORRECTED !
<FormulaDecl> ::= <IdOps> ':' <FormulaName> <Expr>

<Judgement> ::= <SubtypeJudgement> | <ConstantJudgement>

! CORRECTION ! Make Judgements less restrictive by using IdOps instead of IdOp
<SubtypeJudgement> ::= <IdOps> ':' 'JUDGEMENT' <TypeExprList> 'SUBTYPE_OF' <TypeExpr>
            |  'JUDGEMENT' <TypeExprList> 'SUBTYPE_OF' <TypeExpr>
<TypeExprList> ::= <TypeExpr> ',' <TypeExprList> | <TypeExpr>
```

<ConstantJudgement> ::= <IdOps> ':' 'JUDGEMENT' <ConstantReferenceList> 'HAS_TYPE' <TypeExpr>
| 'JUDGEMENT' <ConstantReferenceList> 'HAS_TYPE' <TypeExpr>
<ConstantReferenceList> ::= <ConstantReference> ',' <ConstantReferenceList> | <ConstantReference>

<ConstantReference> ::= Number | <Name> <BindingsAsterisk>

<Conversion> ::= 'CONVERSION' <ConversionGroupList>
<ConversionGroupList> ::= <ConversionGroup> ',' <ConversionGroupList> | <ConversionGroup>
<ConversionGroup> ::= <Name> ':' <TypeExpr>
| <Name>

<Datatype> ::= <InlineDatatype>
| <IdOp> <TheoryFormals> <DatatypeHead> <DatatypeBody>

<DatatypeHead> ::= ':' 'DATATYPE' <DatatypeOption1>
<DatatypeBody> ::= 'BEGIN' <DatatypeOption2> <DatatypeOption3> <DatatypePart> 'END' Id
<DatatypeOption1> ::= 'WITH' 'SUBTYPES' <Ids> |
<DatatypeOption2> ::= <Importing> ';' | <Importing> |
<DatatypeOption3> ::= <AssumingPart> |

<InlineDatatype> ::= <IdOp> <DatatypeHead> <DatatypeBody>

<DatatypePart> ::= <DatatypePartGroup> <DatatypePart>
| <DatatypePartGroup>
<DatatypePartGroup> ::= <Constructor> ':' <IdOp> ':' Id
| <Constructor> ':' <IdOp>

<Constructor> ::= <IdOp> '(' <ConstructorGroupList> ')'
| <IdOp>
<ConstructorGroupList> ::= <ConstructorGroup> ',' <ConstructorGroupList>
| <ConstructorGroup>
<ConstructorGroup> ::= <IdOps> ':' <TypeExpr>

<TypeExpr> ::= <Name>
| <EnumerationType>
| <Subtype>
| <TypeApplication>
| <FunctionType>
| <TupleType>
| <RecordType>

<EnumerationType> ::= '{' <IdOp> '}'
| '{' <IdOp> ',' <IdOps> '}'

<Subtype> ::= '{' <SetBindings2> '|' <Expr> '}'
| '(' <Expr> ')'

<SetBindings2> ::= <IdOp> ':' <TypeExpr> ',' <SetBindings2>
| <IdOp> ':' <TypeExpr> <SetBindings2>
| <IdOp> ':' <TypeExpr>
| <IdOp> ',' <SetBindings2>

```
            |  <IdOp> <SetBindings2>
            |  <IdOp>
            |  <Bindings> ',' <SetBindings2>
            |  <Bindings> <SetBindings2>
            |  <Bindings>

!<SetBinding> ::= <IdOp> ':' <TypeExpr>
!           |  <IdOp>
!           |  <Bindings>

<TypeApplication> ::= <Name> <Arguments>

<FunctionType> ::= <FunctionHeader> '[' <FTTypeGroupList> '->' <TypeExpr> ']'
            |  '[' <FTTypeGroupList> '->' <TypeExpr> ']'
<FunctionHeader> ::= 'FUNCTION'|'ARRAY'

<TupleType> ::= '[' <FTTypeGroupList> ']'
<FTTypeGroupList> ::= <FTTypeGroup> ',' <FTTypeGroupList> | <FTTypeGroup>
<FTTypeGroup> ::= <IdOp> ':' <TypeExpr> | <TypeExpr>

<RecordType> ::= '[#' <FieldDeclsList> '#]'
<FieldDeclsList> ::= <FieldDecls> ',' <FieldDeclsList> | <FieldDecls>

<FieldDecls> ::= <Ids> ':' <TypeExpr>

<Expr> ::= Number
     |  String
     |  <Name>
     |  <Expr> <Arguments>
     |  <Expr> <Binop> <Expr>
     |  '-' <Expr>
     |  <Unaryop> <Expr>
     |  <Expr> '`' Id | <Expr> '`' Number
     |  '(' <ExprList> ')'
     |  '(:' <ExprListAsterisk> ':)'
     |  '[|' <ExprListAsterisk> '|]'
     |  '(#' <AssignmentList> '#)'
     |  <Expr> '::' <TypeExpr>
     |  <IfExpr>
     |  <BindingExpr>
     |  '{' <SetBindings> '|' <Expr> '}'
     |  'LET' <LetBindingList> 'IN' <Expr>
     |  <Expr> 'WHERE' <LetBindingList>
     |  <Expr> 'WITH' '[' <AssignmentList> ']'
     |  'CASES' <Expr> 'OF' <SelectionList> 'ELSE' <Expr> 'ENDCASES'
     |  'CASES' <Expr> 'OF' <SelectionList> 'ENDCASES'
     |  'COND' <CondGroupList> 'ENDCOND'
     |  <TableExpr>

<ExprList> ::= <Expr> ',' <ExprList> | <Expr>
<ExprListAsterisk> ::= <ExprList> |
<AssignmentList> ::= <Assignment> ',' <AssignmentList> | <Assignment>
<LetBindingList> ::= <LetBinding> ',' <LetBindingList> | <LetBinding>
```

```
<SelectionList> ::= <Selection> ',' <SelectionList> | <Selection>
<CondGroupList> ::= <Expr> '->' <Expr> ',' <CondGroupList>
            |  <Expr> '->' <Expr>
            |  'ELSE' '->' <Expr>
<IfExpr> ::= 'IF' <Expr> 'THEN' <Expr> <ElsIfExprGroup> 'ELSE' <Expr> 'ENDIF'
<ElsIfExprGroup> ::= 'ELSIF' <Expr> 'THEN' <Expr> <ElsIfExprGroup> |

<BindingExpr> ::= <BindingOp> <LambdaBindings> ':' <Expr>

<BindingOp> ::= 'LAMBDA' | 'FORALL' | 'EXISTS' | <IdOp> '!'

<LambdaBindings> ::= <LambdaBinding> ',' <LambdaBindings>
            |  <LambdaBinding> <LambdaBindings>
            |  <LambdaBinding>

<LambdaBinding> ::= <IdOp> | <Bindings>

<SetBindings> ::= <SetBinding> ',' <SetBindings>
          |  <SetBinding> <SetBindings>
          |  <SetBinding>
          |  <IdOps>
          |  <IdOps> ':' <TypeExpr>

<SetBinding> ::= <IdOp> ':' <TypeExpr>
         |  <IdOp>
         |  <Bindings>

<Bindings> ::= '(' <BindingList> ')'
<BindingList> ::= <Binding> ',' <BindingList> | <Binding>

<Binding> ::= <TypedId> | '(' <TypedIds> ')'

<Assignment> ::= <AssignArgs> <AssignmentGroup> <Expr>
<AssignmentGroup> ::= ':=' | '|->'

<AssignArgs> ::= Id '!' Number
              |  Id
         |  Number
              |  <AssignArgPlus>
<AssignArgPlus> ::= <AssignArg> <AssignArgPlus>
            |  <AssignArg>

<AssignArg> ::= '(' <ExprList> ')'
              |  '`' Id
              |  '`' Number

<Selection> ::= <IdOp> '(' <IdOps> ')' ':' <Expr>
        |  <IdOp> ':' <Expr>

<TableExpr>  ::=  'TABLE'  <TableExprOption1>  <TableExprOption2>  <TableExprOption3>
<TableEntryPlus> 'ENDTABLE'
<TableExprOption1> ::= <Expr> |
<TableExprOption2> ::= ',' <Expr> |
```

```
<TableExprOption3> ::= <ColHeading> |
<TableEntryPlus> ::= <TableEntry> <TableEntryPlus>
              |  <TableEntry>

<ColHeading> ::= '|[' <Expr> <ColHeadingGroupPlus> ']|'
<ColHeadingGroupPlus> ::= <ColHeadingGroup> <ColHeadingGroupPlus>
                  |  <ColHeadingGroup>
<ColHeadingGroup> ::= '|' <Expr> | '|' 'ELSE'

<TableEntry> ::=  <TableEntryGroupPlus> '||'
<TableEntryGroupPlus> ::= <TableEntryGroup> <TableEntryGroupPlus>
                  | <TableEntryGroup>
<TableEntryGroup> ::= '|' <Expr> | '|' 'ELSE'

<LetBinding> ::= <LetBindingGroup> '=' <Expr>
<LetBindingGroup> ::= <LetBind> | '(' <LetBindList> ')'
<LetBindList> ::= <LetBind> ',' <LetBindList> | <LetBind>

<LetBind> ::= <IdOp> <BindingsAsterisk> ':' <TypeExpr>
         | <IdOp> <BindingsAsterisk>
<BindingsAsterisk> ::= <Bindings> <BindingsAsterisk> |

<Arguments> ::= '(' <ExprList> ')'
                          !<ExprList> is already defined

<TypedIds> ::= <IdOps> ':' <TypeExpr> '|' <Expr>
          | <IdOps> ':' <TypeExpr>
          | <IdOps> '|' <Expr>
          | <IdOps>

<TypedId> ::= <IdOp> ':' <TypeExpr> '|' <Expr>
         | <IdOp> ':' <TypeExpr>
         | <IdOp> '|' <Expr>
         | <IdOp>

<TheoryNames> ::= <TheoryName> ',' <TheoryNames> | <TheoryName>

<TheoryName> ::= Id '@' Id <Actuals>
           | Id '@' Id
           | Id <Actuals>
           | Id

! SHIFT-REDUCE Correction !
<Name> ::= <NameHead> <Actuals> '.' <IdOp>
     |  <NameHead> <Actuals>
     |  <NameHead> '.' <IdOp>
     |  <NameHead>
<NameHead> ::= <IdOp> '@' <IdOp> | <IdOp>

<Actuals> ::= '[' <ActualList> ']'
<ActualList> ::= <Actual> ',' <ActualList> | <Actual>

<Actual> ::= <Expr> | <TypeExpr>
```

```
        |  <Actual> '`' Id
        |  <Actual> '`' Number
        |  <Actual> <Binop> <Expr>
        |  <Actual> <Arguments>
        |  <Actual> '::' <TypeExpr>
        |  <Actual> 'WHERE' <LetBindingList>
        |  <Actual> 'WITH' '[' <AssignmentList> ']'

<IdOps> ::= Id ',' <IdOps> | <Opsym> ',' <IdOps> | <Opsym> | Id

<IdOp> ::= Id | <Opsym>

<Opsym> ::= <BinOp> | <UnaryOp> | <OpsymOthers>
<OpsymOthers> ::= 'IF' | 'TRUE' | 'FALSE' | '[||]'

<BinOp> ::= 'o' | 'IFF' | '<=>' | 'IMPLIES' | '=>' | 'WHEN' | 'OR'
       | '\/' | 'AND' | '/\' | '&' | 'XOR' | 'ANDTHEN' | 'ORELSE'
       | '^' | '+' | '-' | '*' | '/' | '++' | '~' | '**' | '//' | '^^'
       | '|-' | '|=' | '<|' | '|>' | '=' | '/=' | '==' | '<' | '<='
       | '>' | '>=' | '<<' | '>>' | '<<=' | '>>=' | '#' | '@@' | '##'

<UnaryOp> ::= 'NOT' | '~' | '[]' | '<>' | '-'

<FormulaName> ::= 'AXIOM' | 'CHALLENGE' | 'CLAIM' | 'CONJECTURE' | 'COROLLARY'
             | 'FACT' | 'FORMULA' | 'LAW' | 'LEMMA' | 'OBLIGATION'
                 | 'POSTULATE' | 'PROPOSITION' | 'SUBLEMMA' | 'THEOREM'

<Ids> ::= Id ',' <Ids> | Id
Id = {Letter}{IdChar}*

Number = {Number}+


{String Chars} = {Printable} - ["]
String = '"' {String Chars}* '"'


{IdChar} = {AlphaNumeric} + [_] + [?]
```

## APPENDIX B. PVS Proof Scripts in BNF

```
"Name"     = 'Enter the name of the grammar'
"Author"   = 'Enter your name'
"Version"  = 'The version of the grammar and/or language'
"About"    = 'A short description of the grammar'

"Start Symbol" = <Program>

! ------------------------------------------------ Sets

{ID Body}      = {Printable} - [()"|] - {Whitespace}
{String Chars} = {Printable} - ["\]

! ------------------------------------------------ Terminals

Identifier    = {ID Body}+
StringLiteral = '"' ( {String Chars} | '\' {Printable} )* '"'

! ------------------------------------------------ Rules

<Program> ::= <TheoryProofList>

<TheoryProofList> ::= <TheoryProofs> <TheoryProofList>
                    |

<TheoryProofs> ::= '(' '|' Identifier '|' <ProofList> ')'

<ProofList> ::= <Proof> <ProofList>
              |

<Proof> ::= '(' '|' Identifier '|' <ProofBody> ')'

<ProofBody> ::= StringLiteral <SingleCommand> <ManyCommands> 'NIL'

<SingleCommand> ::= '(' Identifier <SubExpr> ')'
                  |  'NIL'

<ManyCommands> ::=  '(' <CommandContainers> ')'
```

```
            |   'NIL'

<CommandContainers> ::= <CommandContainer> <CommandContainers>
                |   <CommandContainer>

<CommandContainer> ::= '(' <ProofBody> ')'

<SubExpr> ::= <Id> <SubExpr>
        |   <ParenExpr> <SubExpr>
        |

<ParenExpr> ::= '(' <SubExpr>')'

<Id> ::= Identifier
    |   StringLiteral
```

## APPENDIX C. Accessing the VAMP Explorer

The VAMP Explorer can be accessed online at:

http://www.natidea.com/projects/VAMPExplorer

This link provides access to some of the preliminary prototypes as well as the latest functional version of the interface.

## REFERENCES


[Al03] G. Alder. Design and Implementation of the JGraph Swing Component. 2003. http://www.jgraph.com, (Accessed July 2005).

[BB+02] C. Berg et. al. Formal Verification of the VAMP Microprocessor: Project Status. Saarland University, February 2002.

[Be00] B.B. Bederson. Fisheye Menus. In *Proceedings of ACM Conference on User Interface Software and Technology (UIST 2000)*, pp. 217-226, November 2000. Demo at http://www.cs.umd.edu/hcil/fisheyemenu/, (Accessed July 2005).

[Be04] S. Beyer. *Putting It All Together – Formal Verification of the VAMP*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, 2004.

[Ber01] Christoph Berg. *Formal Verification of an IEEE floating point adder*. Master's thesis, Saarland University, Saarbrücken, Germany, 2001.

[BJ01] Christoph Berg and Christian Jacobi. Formal Verification of the VAMP Floating Point Unit. In *CHARME 2001*, volume 2144 of LNCS, pp 325-339. Springer, 2001.

[BJK01] Christoph Berg, Christian Jacobi, and Daniel Kröning. Formal Verification of a Basic Circuits Library. *Proc. 9th IASTED International Conference on Applied Informatics*, Innsbruck (AI'2001), pp 252-255. ACTA Press, 2001.

[BKM96] Bishop Brock, Matt Kaufmann and J Moore. ACL2 Theorems about Commercial Microprocessors. In M. Srivas and A. Camilleri (eds.) *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, Springer-Verlag, pp. 275-293, 1996.

[Bra05] TheBrain Technologies Corporation, http://www.thebrain.com/, (Accessed July 2005).

[BSW02] B. Bederson, B. Shneiderman, and M. Wattenberg. Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies. *ACM Transactions on Graphics (TOG)*, 21, (4), October 2002, 833-854.

[Co04] Devin Cook. *Design and Developement of a Grammar Oriented Parsing System*. Master's thesis, Calfornia State University, Sacramento, CA, 2004. http://www.devincook.com/goldparser, (Accessed July 2005).

[CW+00] A. Crapo, L. Waisel, W. Wallace and T. Willemain. Visualization and the process of modeling: a cognitive-theoretic view. *Proc. of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 218 – 226, Boston, MA, 2000.

[Fo01] A.C.J. Fox. An Algebraic Framework for Modeling and Verifying Microprocessors Using HOL. *Technical report No. 512*, University of Cambridge Computer Laboratory, March 2001.

[GCE99] Nahum Gershon, Stuart Card, Stephen G. Eick. Information visualization tutorial. *CHI '99 extended abstracts on Human factors in computing systems*, pp. 149 – 150, May 15-20, 1999, Pittsburgh, Pennsylvania.

[Gvz05] Graphviz – Graph Visualization Software. AT&T. http://www.graphviz.org/, (Accessed July 2005).

[GXL] Ric Holt , Andy Schürr, Susan E. Sim, and Andreas Winter. *GXL – Graph eXchange Language*. http://www.gupro.de/GXL, (Accessed July 2005).

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[Ja02] Jacobi, C., *Formal Verification of a Fully IEEE Compliant Floating Point Unit*, PhD Thesis, University of the Saarland, 2002.

[JS04] J. Jelinek and P. Slavik. XML visualization using tree rewriting. In *Proceedings of the 20th spring conference on Computer graphics*, Pg 65 - 72, Budmerice, Slovakia, 2004

[KG99] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, v.4 n.2, p.123-193, April 1999

[Kin03] J. Kiniry. Formalizing the User's Context to Support User Interfaces for Integrated Mathematical Environments. *Electronic Notes in Theoretical Computer Science*, Volume 103, 3 Nov. 2004, Pages 81-103.

[Kr99] Thomas Kropf. *Introduction to Formal Hardware Verification*. Springer-Verlag, Berlin, Germany, 1999.

[Krö01] D. Kröning. *Formal Verification of Pipelined Microprocessors*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, 2001.

[Mac05] Macromedia – Flash MX 2004. http://www.macromedia.com/software/flash, (Accessed July 2005).

[MP00] S. M. Mueller and W. J. Paul. *Computer Architecture. Complexity and Correctness*. Springer-Verlag, Berlin, Germany, 2000.

[OS03] S. Owre and N. Shankar. The PVS Prelude Library. *CSL Technical Report SRI-CSL-03-01*, SRI International, Menlo Park, CA, March 2003.

[OSR01] S. Owre, N. Shankar, J.M. Rushby and D. Stringer-Calvert. *PVS System Guide*. Version 2.4, SRI International, Menlo Park, CA, November 2001.

[OSR01b] S. Owre, N. Shankar, J.M. Rushby and D. Stringer-Calvert. *PVS Language Reference*. Version 2.4, SRI International, Menlo Park, CA, November 2001.

[PS05] P. Seidel. A Short Introduction to Theorem Proving. Lecture notes, Southern Methodist University, Dallas, Texas, March 2005.

[PS05b] P. Seidel, Theorem Proving. Lecture notes, Southern Methodist University, Dallas, Texas, March 2005.

[SPR02] Jennifer Preece, Yvonne Rogers and Helen Sharp. *Interaction Design*. Wiley, Hoboken, NJ, January 2002.

[Sun05] Sun Microsystems, Inc. Javadoc Tool. http://java.sun.com/j2se/javadoc/, (Accessed July 2005).

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, volume 11 (1), pages 25–33. IBM, 1967.

[Vis05] Visualization at the Human-Computer Interaction Lab. University of Maryland. http://www.cs.umd.edu/hcil/research/visualization.shtml (Accessed July 2005).

[VMP05] Institut für Rechnerarchitektur und Parallelrechner – VAMP. http://www-wjp.cs.uni-sb.de/forschung/projekte/VAMP/downloads.php (Accessed July 2005).

[WB98] Christopher Westphal and Teresa Blaxton. *Data Mining Solutions*. Wiley Computer Publishing, New York, 1998.