Panoptes: An Exploration Tool for Formal Proofs

Panoptes: An Exploration Tool for Formal Proofs

By
Orlin Grigorov, B.A.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

M.Sc. in Computer Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2008) McMaster University
(Computer Science) Hamilton, Ontario

TITLE: Panoptes: An Exploration Tool for Formal Proofs

AUTHOR: Orlin Grigorov
B.A.(American University in Bulgaria)

SUPERVISOR: Dr. William M. Farmer

NUMBER OF PAGES: viii, 113

ABSTRACT

Proof assistants aid the user in proving mathematical theorems by taking care of low-level reasoning details. Their user interfaces often present proof information as text, which becomes increasingly difficult to comprehend as it grows in size. Panoptes is a software tool that enables users to explore graphical representations of the formal proofs produced by the IMPS Interactive Mathematical Proof System. Panoptes automatically displays an IMPS deduction graph as a visual graph that can be easily manipulated by the user. Its facilities include target zooming, floating information boxes, node relabeling, and proper substructure collapsing.

CONTENTS

Abstract iii

1 Introduction 1

1.1 Formalized Mathematics 1

1.2 Proof Assistants 1

1.3 Objectives of the Thesis 2

1.4 Organization of the Thesis 3

2 Background 5

2.1 IMPS 5

2.2 Sequents 5

2.3 Proof Representation 6

2.3.1 Prescriptive Approach 6

2.3.2 Descriptive Approach 6

2.3.3 Deduction Graphs 7

2.3.4 Grounded Nodes 8

3 Problem 9

3.1 Navigating Through the Deduction Graph 9

3.2 Detecting Nodes with Similar Semantics 12

3.3 Avoiding Redundant Work 13

4 Solution **14**
4.1 New Tool 14
4.2 Targeted Users 14
4.3 Who was Panoptes? 15
4.4 List of Synonyms 15

5 Requirements of the System **16**
5.1 Overview 16
5.2 Functional Requirements 16
5.2.1 Visualization of Nodes 17
5.2.2 Naming Conventions 17
5.2.3 Information Boxes 18
5.2.4 Positioning of Nodes 19
5.2.5 Zooming Function 20
5.2.6 Collapsing Parts of the Graph 20
5.2.7 Undoing Operations 21
5.2.8 Helping the User 21
5.3 Nonfunctional Requirements 22
5.3.1 User Characteristics 22
5.3.2 Usability 22
5.3.3 Hardware Considerations 23
5.3.4 Performance Characteristics 23
5.3.5 System Interfacing 25
5.3.6 Portability 25
5.3.7 Management Issues 25

6 Design of the System **26**
6.1 Overview 26
6.2 Revisiting the Functional Requirements 27
6.2.1 Visualization of Nodes 27
6.2.2 Naming Conventions 28
6.2.3 Information Boxes 29
6.2.4 Positioning of Nodes 31
6.2.5 Zooming Function 31
6.2.6 Collapsing Parts of the Graph 31
6.2.6.1 Single Subgoal Chain of Deduction Steps 32

6.2.6.2 Multiple Subgoals Deduction Steps 33
6.2.6.3 Multiple Proof Directions (Branching) 34
6.2.6.4 Cycles 35
6.2.6.5 Converging Branches 36
6.2.6.6 Examining Collapsed Nodes 38
6.2.6.7 Naming of Collapsed Nodes 39
6.3 Integration with IMPS 40
6.3.1 How IMPS Works 40
6.3.2 Actions, Induced by IMPS and Its User Interface 41
6.3.3 Actions, Induced by the User 41
6.3.4 Integrating Panoptes In the Process 41
6.3.4.1 Flow of Commands (Messages) 42
6.3.4.2 Data Flow 43
6.3.5 History of Operations Between Deduction Steps 44
6.4 Structuring the System 44
6.5 Shortcomings of the Design 45

7 Prototype Implementation of the System 46
7.1 Overview 46
7.2 Programming Choices 46
7.2.1 Objective Caml (vs. C/C++) 47
7.2.2 OpenGL 48
7.2.2.1 Three-dimensional Space and Zooming 49
7.2.2.2 User Interface Elements 50
7.2.2.3 Text in OpenGL 50
7.2.2.4 OpenGL Literature 51
7.2.3 Graphviz 52
7.3 Installation and Use 52
7.4 Demonstration 53
7.5 Screenshots 54

8 Related Work 64

9 Future Work 66

10 Conclusion 68

11 Acknowledgments **69**

Appendix A—Module Guide **70**

A.1 Anticipated Changes 70
A.2 Unlikely Changes 71
A.3 Module Decomposition 72
A.4 Description of Modules 72
A.4.1 Keyboard Input Module (external) 73
A.4.2 Mouse Control Module (external) 73
A.4.3 File Read/Write Module (external) 73
A.4.4 Window Initialization Module (external) 73
A.4.5 OpenGL Module (external) 74
A.4.6 Automatic Layout Module (external) 74
A.4.7 Types Module 74
A.4.8 Parser Module 74
A.4.9 Sqn_grabber Module 75
A.4.10 Parser_dot Module 75
A.4.11 TextGL Module 75
A.4.12 Node Module 75
A.4.13 Arrow Module 76
A.4.14 Canvas Module 76
A.4.15 Panoptes Module 76
A.5 Uses Hierarchy 76
A.6 Summary of Requirements 78
A.6.1 List of Functional Requirements 78
A.6.2 List of Non-Functional Requirements 79
A.7 Traceability Matrix 81

Appendix B—Documentation of the Implementation **82**

B.1 Module `Sqn_grabber` 83
B.2 Module `Types` 83
B.3 Module `TextGL` 84
B.4 Module `Parser` 87
B.5 Module `Node` 91
B.6 Module `Arrow` 96
B.7 Module `Parser_dot` 97

B.8 Module `Canvas` 98
B.9 Module `Panoptes` 109

CHAPTER 1

INTRODUCTION

1.1 Formalized Mathematics

In [4], Dr. W. M. Farmer talks about the difference between formalized and conventional mathematics. According to him "formalized mathematics offers greater rigor than conventional mathematics," which is supported by the fact that in formalized mathematics there is more precision in the language of formal logic than there is in the informal (or conventional) mathematical language. Furthermore, it is totally acceptable and possible for formal proofs to be checked by a machine, and also the process of mathematical reasoning can be mechanized. Thus "more opportunity for mechanical support" is offered by formalized mathematics than conventional mathematics can offer. Of course, there are certain drawbacks in formalized mathematics, such as the overwhelming amount of detail that is produced, as well as the fact that not much mathematics of this form has been produced, although there is more and more as time passes. Accessibility problems remain though—ordinary mathematicians cannot benefit from it as much because it is not in a form that they are used to and understand.

1.2 Proof Assistants

When dealing with proofs, a proof assistant (a.k.a. a computer assisted theorem prover) aids the user in the development of a formal proof. It is important to know

that these systems do not fully automate this process, as that is quite ineffective—the search space is too large (often even supercomputers would not be capable of iterating through all of the possibilities in a reasonable amount of time). Also, searches that fail normally provide little useful feedback to the user. Usually, computer assisted theorem provers automatically take care of the low-level reasoning, while the user is given the opportunity to choose the more global direction of the search for a proof.

1.3 Objectives of the Thesis

The user interfaces of proof assistants are mostly text based. Also, formal proofs produced by them are often quite lengthy with many steps and each step produces a significant amount of information. This makes it difficult to manually inspect and successfully follow and understand complete proofs. Furthermore, while developing a proof, the user needs to be careful and know exactly what he or she is doing, for which a strong understanding of the stage and position in the proof is vital for success. It is extremely difficult to comprehend all the information contained in the text of the proof, which is a major reason mathematicians do not choose to use proof assistants in their work. Simply put, because of the text-based approach to transmitting information from the computer to the user, the use of computer assisted proof assistants is not yet a popular technique for producing proofs. Furthermore, these are proofs that are not only checked by a machine, hence guaranteed to be correct, but also potentially easier to create due to the assistance from the system.

This thesis accounts for the inconveniences and inefficiencies of the common text-based user interfaces, which are used in computerized theorem provers nowadays, and then proposes the requirements, design and a sample implementation of a program that takes theorem proving to another level—graphical representation of completed proofs or proofs in progress, where the user has a global overview of the structure, context and information before his or her eyes without too much staring at text and straining to decode the information contained in these proofs. The more global goal is to reduce the fear of using a computer-assisted theorem prover. The user must devote considerable time to finding his or her way around the proof structure instead of concentrating on proving the goals and being able to focus on the mathematical side of the matter at hand.

1.4 Organization of the Thesis

This thesis proposes a new software program that will create a more manageable environment for people to use software theorem provers. As such, the document is divided into several major chapters, each of which deals with a different phase of the development life-cycle of a program. It ends with two appendices, which attempt to further enhance the reader's comprehension of the material covered by this work, and a bibliography list.

Chapter 2 (Background) provides background about the theorem prover, which was chosen for the case study of this thesis. Detailed information on its capabilities and way of proving theorems is given, as well a description of its output format. The latter has a great impact on the requirements and design of the proposed system.

Chapter 3 (Problem) states the major problem areas associated with the text-based delivery of results, and a claim is made about the inefficiency of this approach. The identified problems in this chapter will serve as a basis for gathering requirements and designing the new system in the subsequent chapters.

Chapter 4 (Solution) briefly, but clearly, proposes the new system. Also, it describes the kind of potential users who will be interested in using it.

Chapter 5 (Requirements of the System) provides the requirements of the proposed system. All deemed necessary functionality is described here, and also each requirement is labeled with a short name so that it can be referenced later in the document.

Chapter 6 (Design of the System) establishes the algorithms that are used for implementing the functionalities described in the previous chapter.

Chapter 7 (Prototype Implementation of the System) provides a description of a prototype implementation of the tool. The programming choices are described, along with their advantages and other reasons for making the choices. The reader should also expect to see some techniques that were utilized to make a good working system.

Chapter 8 (Related Work) discusses past work of other people that deals with subject matter similar to this thesis.

At the end, **Chapter 9 (Future Work)** offers a discussion that touches on everything from including and implementing new features up to some very brave and ambitious ideas, such as expanding the software into a fully functional standalone user interface.

Appendix A (Module Guide) contains the module guide of the design, which was created concurrently with the process of gathering the requirements.

Appendix B (Documentation of the Implementation) contains the technical documentation of the implementation of the program, where each module, method and variable is documented.

CHAPTER 2

BACKGROUND

2.1 IMPS

IMPS[1] [6], developed for the MITRE Corporation by W. M. Farmer, J. D. Guttman, and F. J. Thayer, is a proof assistant that aids the user in developing formal proofs, where the high-level reasoning is controlled by the user, while the low-level reasoning is done automatically. These proofs are built within formal theories and are a blend of deduction and calculation.

The meaning of the term "formal proof" is slightly different than the meaning found in most mathematical literature. In IMPS, the steps in a proof can be so large that the proof can start to resemble an informal proof. As such, the formal proofs in IMPS can be clear and understandable by humans. These large steps are possible because IMPS automatically does most of the low-level reasoning through expression simplification routines. Still, a proof in IMPS is as formal as it can get, because all 'invisible' details are checked and verified by the machine.

2.2 Sequents

The goals that are to be proved in IMPS are called *sequents* [6]. Rather than just being a formula to be proved, a sequent is associated in a certain IMPS theory with

[1]IMPS stands for "Interactive Mathematical Proof System."

a pair consisting of a single formula, called an *assertion* A, and a finite set of assumptions, called a *context* Γ. Consequently, each sequent can be written in the form $\Gamma \Rightarrow A$, which means that A follows from the assumptions in Γ. Logically[2], if $\Gamma = \{A_1, \cdots, A_n\}$ then the sequent represents the logical formula $A_1 \wedge \cdots \wedge A_n \supset A$.

2.3 Proof Representation

Usually, in the world of theorem provers, the proofs can be represented in either a prescriptive or descriptive way, and IMPS uses both of these ways.

2.3.1 Prescriptive Approach

The prescriptive approach can be one of two kinds: declarative or procedural. In the declarative way the prescription for a proof might say something like "A follows from B," in which case the system will try to come up with a way of showing that "B implies A." To do that, it might try all available commands (or sequences of commands) to provide a proof of the implication. The method of prescribing a proof in a procedural way is different. In this case, the prescription will say something like "use command 25," and the system will go ahead and apply that command. IMPS uses this method, while other systems (i.e. Mizar [25] and ACL2 [17]) use the declarative way of prescribing proofs. However, the declarative way of prescribing proofs has one very big disadvantage—to make it possible to execute the prescription, the steps must be relatively small and the proof can thus become very tedious to manually inspect. The only way of eliminating this disadvantage is if a very powerful theorem prover is used, which can recognize big steps and have available commands for them, so that the procedural-prescriptive proof script can contain relatively big steps. Despite all of that, the declarative way is liked more by people who do not want to understand the intimacies of theorem provers, but all they need to have is the knowledge of how they can take steps which are recognized by the system. That being said, the users do not need to learn the available commands as in the procedural way.

2.3.2 Descriptive Approach

In the descriptive approach of proof presentation, the proof object is shown directly, which is analogous to saying something like "this is the proof object." IMPS calls

[2]The logic assumed is propositional logic.

these proof objects "deduction graphs," and in the current IMPS user interface, these graphs are laid out as if they were trees [5].

2.3.3 Deduction Graphs

A deduction graph has two kinds of nodes: one that is used to represent sequents and is called a *sequent node*, and one to represent inferences and is called an *inference node*. Refer to Figure 2.1, where i is an inference node, and $H_1, \cdots, H_n$ and C are sequent nodes.

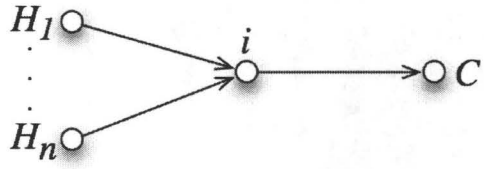

Figure 2.1: An inference node with associated sequent nodes.

As is shown in the diagram, the inference node i forms a logical relationship between the sequent nodes it is connected to: if each of the sequent nodes $H_1, \cdots, H_n$ are valid, then the sequent node C is valid. It also contains the mathematical justification, called the *inference rule*, which justifies this relationship between the sequent nodes. The way a proof is developed in IMPS, though, is backwards in the sense that the above statement can be read and understood as *"the validity of the sequent node C can be reduced to the validity of $H_1, \cdots, H_n$"* [7].

Consequently, the deduction graphs in IMPS are *bipartite* graphs: there are no edges between nodes of the same kind. That is, a *sequent* node will never be connected to another *sequent* node, and an *inference* node will never be connected to another *inference* node.

As can be seen in the figure above, the deduction graph is also a directed graph: a directed edge from a *sequent* node to an *inference* node denotes that the *sequent* node is a hypothesis for the inference represented by the *inference* node. Thus, an *inference* node can have many directed edges pointing at it from many *sequent* nodes, because an inference may have more than one hypothesis, though this is not the case for directed edges that originate from *inference* nodes. There can be only one such

link between an *inference* node and a *sequent* node, the latter representing the conclusion of the inference. Last but not least, it is worth mentioning that the deduction graphs may contain cycles, and having the properties of bipartite graphs, it follows that a cycle will always contain an even number of edges.

2.3.4 Grounded Nodes

Referring back to Figure 2.1, in the previous section it was explained that the sequent node C is guaranteed to be valid if all of the sequent nodes $H_1, \cdots, H_n$, representing the hypotheses of the connecting inference node i, are valid. Also, if the inference node i had no hypotheses at all, the conclusion node C would be automatically valid as well.

In IMPS, sequent nodes that are valid are said to be *grounded sequent nodes.* Furthermore, the inference nodes whose hypotheses are all grounded sequent nodes are said to be *grounded inference nodes.* A proof of a theorem in IMPS is completed when the sequent node that contains the sequent representing the theorem (the starting sequent node) becomes a grounded sequent node.

CHAPTER 3

PROBLEM

This chapter presents the major limitations, which are not covered by the functionality of the IMPS user interface. It is important to understand that these limitations are in no way due to poor design of the user interface (in fact, the IMPS user interface is a product of deep understanding of the problem and it takes full advantage of the abilities of the platform on which it is based) but are a consequence of the nature of Emacs, the software system on which it is based. Namely, Emacs is entirely based on dealing with information in the form of text. This chapter explains the limitations associated with that, and the next chapter (Chapter 4) proposes a new system that attempts to reduce or completely remove these obstacles when the user is developing a formal proof in a computer-assisted theorem prover.

3.1 Navigating Through the Deduction Graph

Sometimes a proof can become quite lengthy, so the deduction graph may contain hundreds of nodes and edges. The current IMPS user interface runs in the text-based Emacs environment, and the deduction graph is represented by a lisp-style, parenthesis-intensive format, suitable for representing trees in text form. Cycles in the graph are taken care of by simply repeating a node name with an indication that the node appeared earlier; it is up to the user to find the node and to understand the structure of the graph. Of course, the IMPS user interface is designed to take complete advantage of all available features that Emacs can offer, such as coloring,

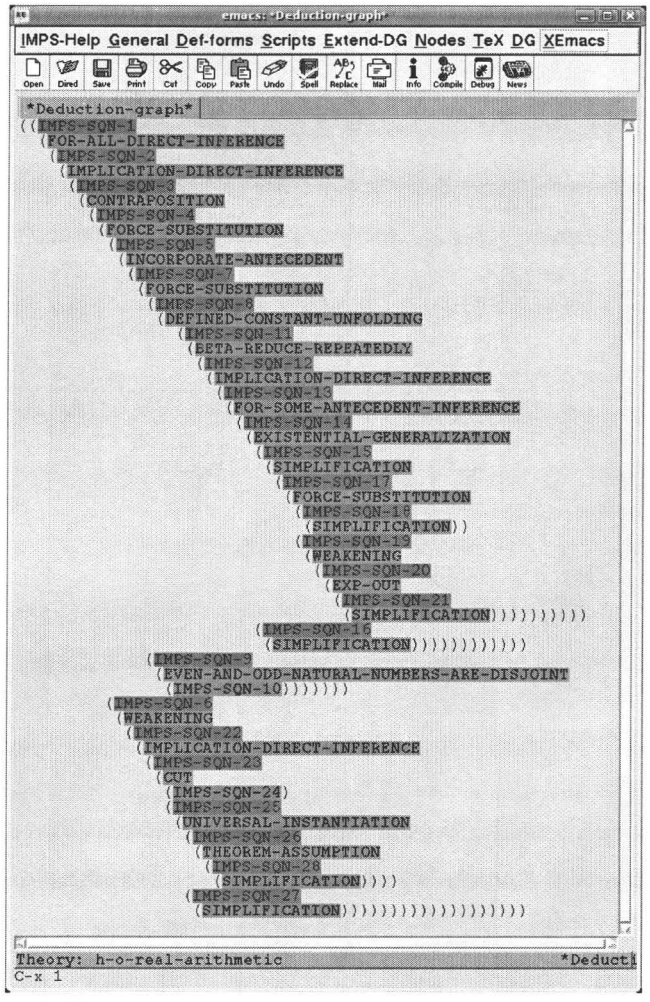

Figure 3.1: Current text based representation of a deduction graph

indentation, searching and parenthesis matching. Still, because of the fact that everything is text, the user has to acquire substantial skill in working with Emacs before they can actually become more comfortable and productive in using the user interface. Although knowledge and Emacs experience can serve the user very well when the proofs are relatively short, when creating more elaborate and serious proofs, the deduction graph will continue to grow, including hundreds of nodes in a very short time. In that case, it is no longer possible, even with dedication and extreme effort, to understand and analyze the proof structure well enough to either comprehend the direction of the proof or to make an informed judgement on the next steps to take in the proof. Very often the proof can also take a direction that will not lead to completion, so the user should be able to detect that in time and either start with

a different approach from an earlier point (or even the beginning) without wasting effort and time working on something that will not lead to success.

Figure 3.1 shows an example of a very short proof session with IMPS. The screenshot shows the Emacs buffer, which displays the current deduction graph that represents the proof of the following theorem: if a number x, which is equal to a certain natural number y squared, is even, then y is even as well. Figure 3.2 shows the structure of the same deduction graph in graphical form, where each circle represents a sequent node and each square represents an inference node.

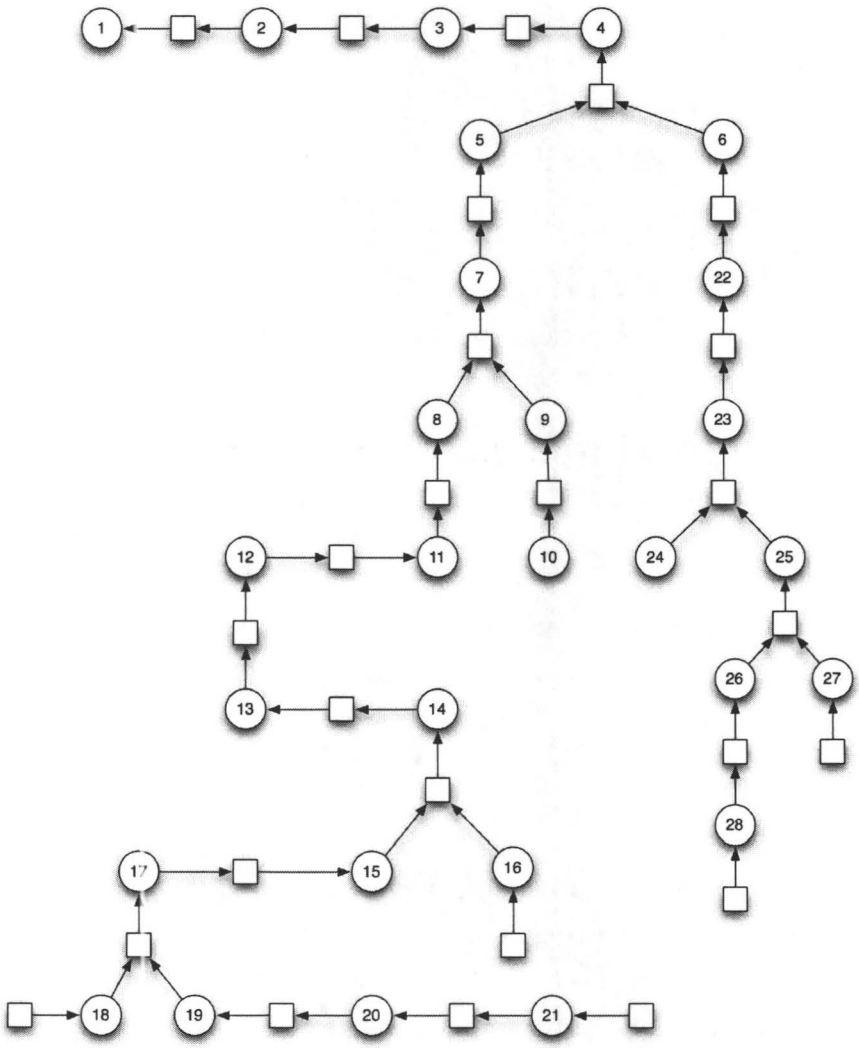

Figure 3.2: Representation of a deduction graph in graphical form

It is important to understand that these figures give an example of a very small deduction graph, which consists of only 28 sequent nodes. Imagine a proof, which contains 20 times as many nodes. Then it would be extremely difficult to navigate through the graph and understand the structure, and notice parts of the graph which look similar. The latter could potentially lead the user towards realizing how certain parts can be proved in similar ways. Additionally, even though IMPS provides the useful functionality of automatically focusing on unproven nodes, it does not provide the user with a way to put nodes side to side to decide which one he or she wants to work on. For example, consider the situation where the user applies induction to a sequent. This command will create additional nodes that need to be proven, such as the base for the induction and the induction step. If the graph contains other unproven goals, the IMPS user interface can only offer support for toggling between all unproven goals, without clearly indicating how and why the unproven nodes appeared. It can be argued that this would be clear from visually inspecting the information in the sequent, but this will cease to be easy to determine with large proofs, where, for example, induction was applied a few times already. Another way of dealing with this is to visually inspect the deduction graph that is offered in the Emacs buffer, but the inconvenience of that must be clear by now from the example given above—the user will become disoriented almost immediately.

3.2 Detecting Nodes with Similar Semantics

Sometimes a proof can reach a sequent, which contains a certain expression either in its context or in its assertion, and it may be similar in meaning with an expression, which was dealt with before at an earlier step in the proof or in a different proof branch of the same goal. For example, consider the following two expressions:

1. "2 / 3"
2. "[2/3]"

IMPS understands the first expression as having three separate constants: "/", "2", and "3". It treats these three constants as an application of "/" to "2" and "3" to form the expression. The second expression, though, is a constant itself. It represents the rational number that is obtained by dividing 2 by 3. In the end, both expressions mean the same thing, but IMPS does not detect and respect their equivalence. To do so, the user would probably need to request simplification or some other relevant

manipulation. Therefore, if the user has the means to put these two pieces of information simultaneously in sight, the human brain can immediately spot that the two nodes might be dealing with the same thing, either in part or fully, which might prevent the user from wasting time and effort on working on something, which is seemingly different, but in reality the same. Such visualization and comparison of the information, contained in the deduction graph with respect to easy observation and exploration, is currently impossible with the existing facilities.

3.3 Avoiding Redundant Work

Another situation arises when there are different sequents with the same assertion[1], but different assumptions[2]. If one of them is proven, then it could be the case that certain assumptions in another object are not necessary. For example, if we have already proven $\{H_1, H_2, H_3\} \Longrightarrow C$, where H_1, H_2 and H_3 are the hypotheses and C is the conclusion, and we have already seen that to prove C we did not use H_3, then later in the proof if we meet a sequent $\{H_1, H_2, H_4\} \Longrightarrow C$ that needs to be proven, IMPS will not automatically detect our discovery that for the proof of C only H_1 and H_2 are needed. Consequently, unless we explicitly remember the latter fact, we will end up trying to discover a proof for C once again, this time based on the old hypotheses and the new ones.

It is worth noting, though, that in the previous example we could use a special command to toss out H_3, in which case IMPS *will* detect automatically the similarity later and will ground the new sequent. The problem with this approach is that tossing out assumptions is dangerous, as later in the proof the user may reach a point where it turns out that these assumptions are actually needed. In that case they will have to redo the proof by adding back the discarded hypotheses.

[1] Assertion = conclusion.
[2] Assumption = hypothesis.

CHAPTER 4

SOLUTION

4.1 New Tool

The objective of this project is to create a tool named Panoptes, which will serve as an add-on to the current IMPS user interface. The purpose of this tool is to allow the user to explore IMPS deduction graphs in a new and more natural way. The main goal is to make it easy to comprehend visually the information contained in the deduction graphs, as well as to give freedom to the user to manipulate and change how they see the information in these structures as easily as possible.

Furthermore, the core of the tool is to be as generic as possible, so that it can be ported to other systems for computer assisted theorem proving by changing a small part of the design and implementation of the tool (possibly just the module that is responsible for retrieving the information from the proof assistant, see Appendix A).

4.2 Targeted Users

The primary target group of users is both mathematicians, who are interactively using IMPS or another proof assistant to search for proofs of theorems while developing theories, as well as computer scientists and software engineers, whose goal is to produce proofs for the correctness of software. Also, the tool should reduce the intimidation novice users, such as students, experience upon their first clash with the

world of formalized mathematics and proof assistants.

4.3 Who was Panoptes?

In Greek mythology, the brother of the nymph Io was a giant with a hundred eyes. His name was Argus, but he was also known as Argus Panoptes, where the epithet means "the all-seeing." Argus was a very effective watchman because his eyes would never go to sleep together at the same time. Thus, having a number of eyes open and watching at all times, he could see everything, and being a giant, he was looking from above [28]. The relation to the proposed *Exploration Tool for Formal Proofs* is that the goal is to make the latter as powerful as Argus Panoptes—that is, to give the best possible overview of large deduction graphs.

4.4 List of Synonyms

The following list provides words and phrases, which are used throughout this paper to denote the same things, and therefore they are used interchangeably:

- Grounded = valid = true = proved = guaranteed;
- Inference = deduction operation (or step);
- Sequent = goal;
- A leaf = a sequent node with no further breakdown, could be grounded or ungrounded;
- Hypothesis = assumption = premise;
- Assertion = conclusion.

CHAPTER 5

REQUIREMENTS OF THE SYSTEM

5.1 Overview

This chapter is divided in two main sections: "Functional Requirements" (5.2) and "Nonfunctional Requirements" (5.3). The first section describes the functionality that Panoptes should provide and each individual requirement is listed and described. By reading this section the reader should expect to acquire a clear idea of what the program does and what functions are available for the user to select. The second section concentrates on issues that need to be addressed by the design and implementation to increase the "quality" of the system rather than what it does. Again, each nonfunctional requirement is listed and labeled, so that it can be referenced later in the paper.

5.2 Functional Requirements

The functional requirements are divided into a few separate groups that provide related goals, or actions that deal with similar matter. There is a separate subsection for each such group.

5.2.1 Visualization of Nodes

This section describes the requirements for the graphical visualization of the nodes on the screen. Also, it explains the reasoning behind some requested features and how they will contribute towards bringing Panoptes closer to the ultimate goal of making the use of IMPS easier (see Section 4.1).

First and foremost, Panoptes should display the graph on the screen in graphical form. This requirement is apparent since it is the actual essence of the system, but it is listed below for the sake of being complete.

☐ **Requirement 1.** *The deduction graph should be displayed in graphical form on the screen.*

The deduction graphs produced by IMPS are bipartite graphs, and as such there are two different kinds of nodes—*sequent nodes* and *inference nodes*, which were described in the previous chapters. The visualization should make it easy for the user to clearly differentiate visually between the two kinds without need of any additional action (such as keystrokes).

☐ **Requirement 2.** *The two kinds of nodes should be visually different when drawn on the screen.*

Also, Panoptes should automatically detect and visually mark 'special' nodes. These comprise grounded nodes, repeated nodes (nodes that complete a cycle or merge proof directions), collapsed nodes (described later in Subsection 5.2.6), as well as nodes which have never been worked on.

☐ **Requirement 3.** *Special nodes should be visually marked to stand out from the rest.*

5.2.2 Naming Conventions

Each node in a deduction graph created by IMPS is unique with respect to all the others. Each sequent node represents a unique sequent. In the case of the inference nodes, there might be more than one node that represents applications of the same inference rule, but the nodes are still unique when viewed in combination with their hypotheses and conclusion sequent nodes. Therefore, Panoptes should provide unique

names (or labels) for each node[1].

IMPS automatically assigns a text name to each newly created sequent node in the deduction graph. Instead of generating new unique names, Panoptes should use in its visualization the same names generated by IMPS. This has the advantage of facilitating the user's ability to cross-reference nodes displayed in Panoptes with the same nodes in the IMPS user interface, where the user applies the proof commands.

□ **Requirement 4.** *Sequent nodes should be labeled on the screen with the names generated by IMPS.*

As mentioned above, each inference node represents an application of an inference rule and there is a chance that a proof or a proof attempt, represented by a deduction graph, uses certain inference rules more than once. Still, to adhere to the goal of having unique nodes in the graph, Panoptes should implement a mechanism for naming the inference nodes in a way that will both preserve the comprehension of what the node represents by just reading its name as well as the uniqueness of that name in respect to all other nodes in the graph.

□ **Requirement 5.** *At initial startup of Panoptes, the name of an inference node should be unique but still indicate the inference rule it is associated with.*

However intelligent the automatic naming algorithm of Panoptes and IMPS could be, there will be cases when the user might prefer to have even more meaningful names of particular nodes of interest. That is why the user should be able to rename such nodes with names according to his or her preference. This will improve the user's experience by making it easier to find his or her way around the graph by a simple glance rather than by examining the contents of the nodes (see Subsection 6.4 below for more information on examining the contents of a node).

□ **Requirement 6.** *The user should be able to rename the labels of nodes.*

5.2.3 Information Boxes

As mentioned in the previous section, every single node carries information. Inference nodes contain the inference rules that generated the represented inferences, and

[1]Notice that when a node is syntactically equivalent to another node, then the deduction graph will contain a cycle rather than having the same node twice.

sequent nodes represent sequents, each of which consists of a number of assumptions and an assertion. In the IMPS user interface the sequent is textually described in the following format:

H_1;
...
H_n;
=>
C;

where $H_1 \cdots H_n$ for $0 \leq n$ represent the assumptions (hypotheses), and C represents the assertion (conclusion) of the sequent. Altogether, this information represents the logical formula $H_1 \wedge \cdots \wedge H_n \supset C$.

It is only logical that the user should be able to examine this detailed information that is associated with a node. A feature of this function is to have a visual link between the display of this information and the node it is associated with, which will prove beneficial for the user's experience.

□ **Requirement 7.** *The user should be able to examine the information associated with sequent and inference nodes without losing track of which node this information is associated with.*

5.2.4 Positioning of Nodes

IMPS stores the deduction graph in a text format (see Figure 3.1 on page 10), which contains information only about the contents and structure of the graph. To provide a visualization of that graph, Panoptes should implement a mechanism for generating position coordinates for each single component of the graph. Furthermore, the generated layout should manage to fit the whole graph in the screen space provided by the system and should also minimize the crossing of edges as much as possible[2].

□ **Requirement 8.** *Upon startup, the program should generate a layout, which fits the whole graph on the screen. The number of edges crossing each other should be minimized.*

[2]Notice that fitting the deduction graph in the screen for big proofs might make it unreadable. That is why the requirements that follow define suitable functionalities to deal with this issue.

In addition, the user should be able to drag and drop components of the graph to either improve or modify the layout according to their preference.

□ **Requirement 9.** *The user should be able to easily rearrange the layout of arbitrary parts of the graph manually.*

5.2.5 Zooming Function

In the case when a deduction graph consists of a large number of nodes, fitting that graph on the screen will make the nodes appear very small especially if the available screen space is relatively small compared to the number of nodes. That is why the user should be capable of easily zooming in and out on parts of the graph to examine them as well as comprehend the structure of the graph.

□ **Requirement 10.** *The program should provide an easy-to-use zooming function.*

5.2.6 Collapsing Parts of the Graph

Some of the complicated commands that IMPS provides produce a large number of nodes when used in a deduction step. Since proofs and proof attempts consist of numerous commands that are applied to break goals into simpler subgoals, the number of nodes in a deduction graph rapidly increases. Consequently, the graph becomes more and more cluttered with information, which may lead to user disorientation and an inability to decide the next proof step.

To make it easy to solve this problem, Panoptes should provide a function to hide selected parts of the graph that might be deemed not important for the process of reasoning about the choice of the next proof command to apply. This "collapsing" function should be safe to execute in respect to preserving the properties of the graph (recall that being a bipartite graph, the deduction graph has two and only two kinds of nodes, which are in turn additionally bound by certain rules described in the Introduction and Background chapters).

□ **Requirement 11.** *Panoptes should provide a function for collapsing deduction graph substructures without changing their meaning.*

Collapsed parts of the graph should not be simply erased by the collapsing function, but rather stored in some manner. Furthermore, the user should be capable of ex-

amining the contents of a collapsed object to understand what part of the graph it encapsulates.

□ **Requirement 12.** *The user should be able to examine the contents of objects that represent collapsed parts of the graph.*

To reduce the need for mandatory examination of the detailed content of these objects, Panoptes should provide a mechanism for generating meaningful short labels for each such object. Each label should be good enough to provide a clear hint to the user regarding the collapsed part of the graph.

□ **Requirement 13.** *Objects, representing collapsed parts of the graph, should have labels for easy identification of the information they contain.*

5.2.7 Undoing Operations

As shown so far, Panoptes should offer a range of operations that the user can apply to the graph. This enables the user to completely transform the appearance of the deduction graph on the screen with respect to the initial visualization that the system provides. It is important that Panoptes keeps a comprehensive history of the operations applied to a deduction graph at all times to allow the user to revert back to an earlier arrangement of the graph on the screen when needed.

□ **Requirement 14.** *Each user induced operation on the visualization of the deduction graph should be reversible.*[3]

5.2.8 Helping the User

A brief help screen should be available during runtime, which can be called by the user for reference to all available commands and their respective shortcuts (keystrokes).

□ **Requirement 15.** *A help screen with available commands and ways to execute them should be present for the user.*

[3]The depth of reversals should be unlimited, so that the user can completely restore the initial graph layout given that they reverse every single operation on the graph that was made up to that point.

5.3 Nonfunctional Requirements

The nonfunctional requirements describe the desired quality of the system, rather than its functionality. The section is divided into a few major categories, and each requirement is listed and explained.

5.3.1 User Characteristics

Section 4.2 already described the typical users of the system. Additionally, the users should be familiar with the services provided by the proof assistant. Before using the system, they should already know how to develop proofs, and should have a complete understanding of the entities that comprise a deduction graph. The users should refer to Chapter 2 to familiarize themselves with the terms used in this document. It will allow them to maximally benefit from the aid the system provides in the process of developing formal proofs.

5.3.2 Usability

The main objective of Panoptes is to make it easier for people to use theorem provers. As such, it must be easy to learn to use, so its user interface should be as straightforward as possible. That is why the user should be able to invoke operations with a mere keystroke, rather than a combination or a sequence of keystrokes. In addition, the user should have an easy and clear understanding about why he or she wants to invoke a certain operation—in other words, each function that the user calls should have a predictable result.

□ **Nonfunctional Requirement 1.** *The system should be easy to learn for users already familiar with the proof assistant.*

□ **Nonfunctional Requirement 2.** *The system should be easy to use for users already familiar with the proof assistant—every operation should be easy to invoke and should have a predictable result.*

Additionally, the system should automatically protect the user from making mistakes, such as attempting to execute actions on irrelevant parts of the graph.

□ **Nonfunctional Requirement 3.** *The system should protect the user from attempting invalid operations on the graph.*

There is no particular need for a separate User's Guide document.[4] Furthermore, Functional Requirement 15 on page 21 requires the system to provide a help screen with all available commands to the user when needed.

Overall, the usability of the system should indicate that any proof development task takes less time to do by using the aid of the system as compared to using only the proof assistant. The larger the task at hand, the more time should be "saved" by choosing to use the system.

5.3.3 Hardware Considerations

The computer system running Panoptes should be equipped with a graphics card that is capable of rendering 3D graphics.[5] Thus, Panoptes should be running smoothly without burdening the user with unnecessary lags and delays. Furthermore, the dedicated graphical processing unit (GPU) of the graphics card should be used for all graphics-related computations, which should in turn decrease the workload put on the central processing unit (CPU). The latter would be thus available for other work (such as that done by the IMPS reasoning engine).

□ **Nonfunctional Requirement 4.** *The system should minimize the workload on the Central Processing Unit (CPU) by having the Graphical Processing Unit (GPU) do as many presentation routine calculations as possible.*

5.3.4 Performance Characteristics

The design and implementation should take into account the fact that the main objective of Panoptes is to aid the user in developing formal proofs, which is a complicated and often difficult process. That is why the system should be very responsive in displaying the result generated by the user operations on the deduction graph. Additionally, good foundations for speed and smoothness of the graphics are imperative as the Department of Computing and Software at McMaster University will soon be equipped with 24 units of 30-inch Apple Cinema HD™ displays[6], each capable of 2560 x 1600 pixels resolution. The screens will be combined (see Figure 5.1) to construct two large units, each consisting of 6 screens (2 x 3 screens, 7,680 x 3,200 pixels

[4] It is assumed that the user already knows how to use IMPS.

[5] Most graphic cards that are sold nowadays are equipped with capabilities to render 3D graphics.

[6] Exact numbers are subject to change.

resolution total), and one huge unit, made of 12 screens (3 x 4 screens, 10,240 x 4,800 pixels resolution total). Even though these large constructions will be controlled by adequately powerful video devices, the design and implementation of Panoptes should be optimized enough to ensure smooth graphics and operations without the need of super powerful graphical devices.

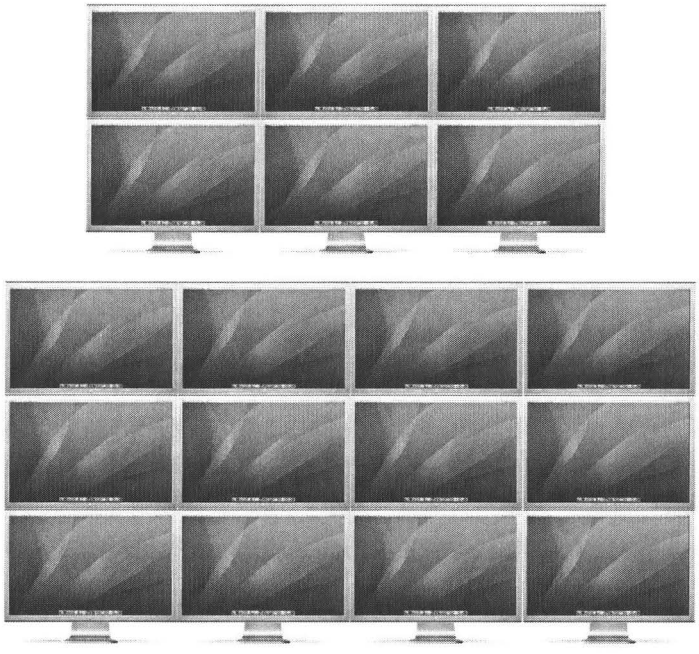

Figure 5.1: Stacked 30-inch Apple Cinema HD displays

Additionally, Panoptes should be able to handle deduction graphs containing many hundreds of nodes. Controlling such large structures involves accounting for a large amount of detail and when it comes to visualization there will be many computations to be done. Therefore, good design and implementation algorithms should be created accordingly.

□ **Nonfunctional Requirement 5.** *Algorithms for the computation of the screen positions of nodes, edges, and other components of the deduction graph should be chosen carefully to ensure smooth runtime performance*[7].

[7]A good measurement would be frames per second (FPS). A good target FPS would be 25 or more, which is more than the TV standard for NTFS motion videos.

5.3.5 System Interfacing

Even though IMPS is the selected system for this study, the design and implementation should be created with the future goal in mind of porting Panoptes to other theorem provers. For that matter, the parts of the system responsible for accepting and processing the input data should be separated from the rest of the system. This will make it easy to modify them to accommodate other proof assistants.

□ **Nonfunctional Requirement 6.** *The parts of the system that are responsible for accepting and processing input*[8] *from the proof assistant should be easily modifiable.*

5.3.6 Portability

Panoptes should compile and be fully operational under all systems that currently support IMPS. For now these are Linux, Mac OS X, and other versions of Unix.

□ **Nonfunctional Requirement 7.** *The system should be compilable and able to run on Unix-style operating systems, particularly Linux and Mac OS X.*

Additionally, Panoptes should be easily modifiable to compile and run on large multiscreen displays (see Section 5.3.4).

□ **Nonfunctional Requirement 8.** *The system run on large screens with little or no modifications.*

5.3.7 Management Issues

Preferably a single installation script should be provided that will compile and prepare the tool for use. The effort of finding and installing external libraries that are needed by the program should be minimized.

□ **Nonfunctional Requirement 9.** *Installation should be performed by simply executing an installation script.*

[8] "Input" refers to the data provided by the client proof assistant. In the current study, this is IMPS.

CHAPTER 6

DESIGN OF THE SYSTEM

6.1 Overview

Creation of Panoptes did not follow any of the standard software development models. It began as a simple idea, which was immediately implemented into a simple working program. From there on, new requirements were added or existing ones modified or refined, and each time that would happen the implementation of the prototype (explained in Chapter 7) was modified or expanded. This process of software development, although known as the "Evolutionary Prototyping Method of Software Development," dangerously approaches the idea behind the term *ad hoc programming* since gathering requirements, design and implementation were three processes happening at the same time, thus resembling one process altogether. However, in [22] Dr. Parnas extensively justifies the statement that as long as the project is documented in a rational way, this should not be a problem.

This chapter does not constitute a design in the format that the reader would expect—there are no class diagrams, object diagrams or other similar design elements. Rather than that, the chapter establishes procedures and algorithms for achieving in the best possible way the functionality of the system listed in the previous Chapter 5. Also, the chapter explains the practical side of the design, such as the methods and principles used and it mentions some possible shortcomings. As such, this chapter is intended for readers interested in learning how the system accomplishes its functionality, while

a reader, who is interested in directly modifying the design or understanding Panoptes as a complete system, should refer to Appendix A, which provides the Module Guide of this design. The appendix shows, explains, and attempts to justify the module decomposition of Panoptes.[1]

6.2 Revisiting the Functional Requirements

This section provides the design decisions made on the functional requirements stated in Chapter 5.

6.2.1 Visualization of Nodes

To be consistent with the literature already written about IMPS, we will adhere to some notational conventions for our examples.

A circle will be used to denote a sequent node and a rectangle (or a square) will be used to denote an inference node in the graph. Furthermore, all nodes are to be connected with arrows, whose direction leads ultimately to the main goal (in the normal case). Figure 6.1 shows a sample deduction graph, which demonstrates this idea.

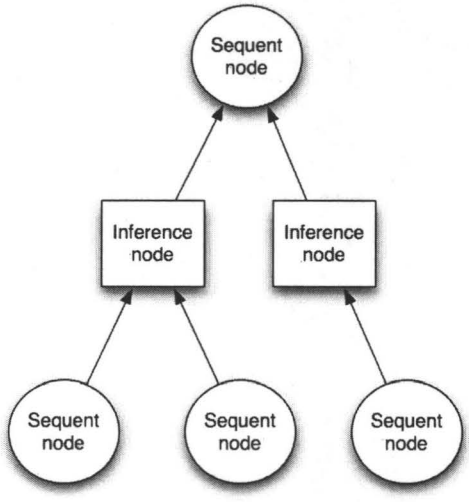

Figure 6.1: Shapes and arrows

[1]Rather than refactoring the source code to obtain the module structure, the latter was obtained by carefully analyzing the source code.

In addition, the requirements call for a visual distinction between special nodes. The best way to accomplish this is by utilizing the power of color. To be consistent with the IMPS user interface, grounded nodes are to be colored in green. Also, repeated nodes are to be colored in brown, and collapsed nodes are to be colored in purple despite the fact that these are not colored in the text version provided by the IMPS user interface.

6.2.2 Naming Conventions

In IMPS the names of the sequent nodes take the form SQN-# where # is a number that shows the order in which the sequent nodes were created. In other words, if $m < n$, then SQN-m was created earlier in the deduction process than SQN-n. Note that generally the distance from the main goal cannot be judged by this number.

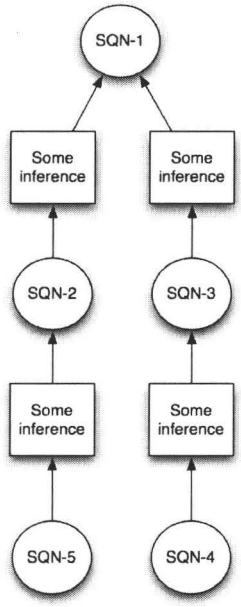

Figure 6.2: Numbers in sequent nodes do not represent depth.

Consider Figure 6.2. Analyzing the graph, the user started with SQN-1 and after applying a command the goal was broken down to SQN-2. Then the user went back to SQN-1 and tried another command, which in turn produced SQN-3. Continuing with the latter, SQN-4 was produced. The user then realized that they would like to continue working on the former starting point of the proof, so he or she applied a command to SQN-2 that produced SQN-5. It is apparent that IMPS names the

nodes in the order they were created, and SQN-5 and SQN-4 are in the same depth level, although in reality $5 > 4$.

The requirements chapter calls for unique names of all nodes, including the inference nodes. Furthermore, it was mentioned that sometimes the same inference rules are used in different proof steps, thus leaving nodes with identical names (the name of an inference node is assigned by IMPS according to the inference rule that is applied). A way to achieve uniqueness is to insert a number at the end of the name, which can also be used to distinguish the order in which the commands were executed. Consider Figure 6.3.

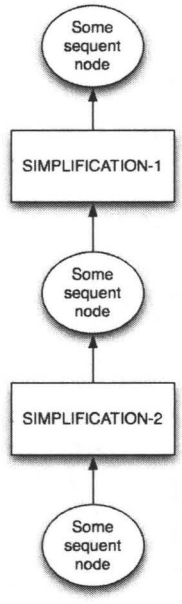

Figure 6.3: Numbering of inference nodes.

The figure provides an example, in which the same command is used more than once in the proof, and how Panoptes should deal with naming the inference nodes.

6.2.3 Information Boxes

The Requirements chapter already showed the kind and format of the information that is represented by the sequent nodes. Also, it was mentioned that there is information associated with the inference nodes as well. That information can be retrieved

from the digital copy of the IMPS User's Manual [7], which provides a detailed description of each possible inference step. Along with that, it also contains a short verbal description of each inference, which is compact enough to be displayed by the information boxes. For instance, the command **cut-with-single-formula** is verbally described in a concise manner as follows:

"This command allows you to add a new assumption to the context of the given sequent. Of course, you are required to show separately that the new assumption is a consequence of the context."

The actual procedure of opening and examining a node starts with the user selecting a node, after which he or she should be able to invoke an "open info box" operation. As a result, a rectangle with information should appear on the screen connected to the node to which it is associated. Simple labeling of the name of the node in the information box is not enough, especially since we are striving to make the deduction graph more visually comprehensive. Therefore the box should be connected by some kind of dashed line to the node of interest. Consider Figure 6.4 for an example of this idea.

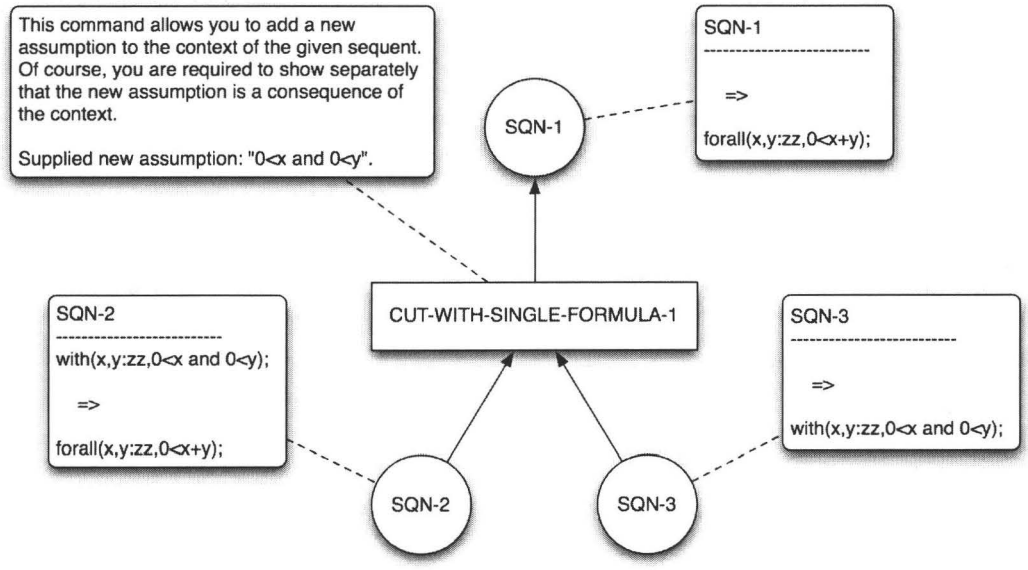

Figure 6.4: Information boxes.

Furthermore, the user should be able to scale and reposition the information boxes

similarly to the manipulation functions provided for the nodes of the graph.

6.2.4 Positioning of Nodes

Since generating an optimal layout is almost a whole science by itself and a lot of research has been conducted on the topic, it is suggestive that a third-party library is used to accomplish the task. See the Implementation chapter (7) for detailed information about the chosen external layout generator.

Due to the hard and complicated task of generating the best possible layout automatically as well as the fact that the user might want to rearrange the graph according to personal preference, the requirements called for functionality for manual rearranging of the graph. This should be accomplished by an easy drag-and-drop capability on the nodes of the graph, while the program automatically protects the connections (the arrows) between the nodes.

6.2.5 Zooming Function

A good zooming design will allow the user not only to zoom in or out easily, but will provide more elaborate mechanisms. One such mechanism is that the zooming is executed on a certain section of the graph rather than on the center of the image. For example, a zoom-in operation over a certain node should simultaneously make the graph bigger and shift the enlarged picture to keep the focus on the selected node at all times. This way, the user will be able to quickly "look into" the part of the graph of interest instead of just zooming in and then shifting the graph until that part becomes visible. That latter effect could also be very confusing, because the user might get lost, especially since he or she is not able to see the context of the graph. See the Implementation chapter (7) for a detailed explanation of an innovative trick used to accomplish this function.

6.2.6 Collapsing Parts of the Graph

This section will define how collapsing should be done to preserve the properties of the graph. The section starts with defining the shapes, used in the diagrams that follow.

A star (see Figure 6.5) will represent a "bag," which contains the information that is collapsed. Furthermore, this star will exhibit the same properties as an inference node, that is, it may have more than one child, but one and only one parent.

Another shape that will be used in the figures in the subsequent subsections is the cloud shape (see Figure 6.6). It will be used to represent collapsed/reduced cycles in the deduction graphs, which are described later in this chapter. The cloud shape will represent a node that exhibits the same properties as a sequent node.

Figure 6.5: A star shape hides parts of the graph and exhibits the same properties as an inference node.

Figure 6.6: A cloud shape hides cycles and exhibits the same properties as a sequent node.

6.2.6.1 Single Subgoal Chain of Deduction Steps

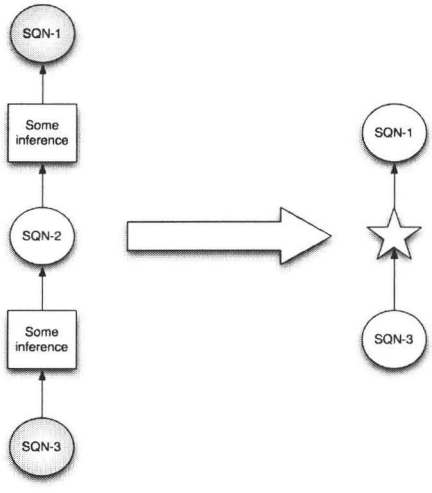

Figure 6.7: Collapsing a chain of deductions that produces a single subgoal.

In Figure 6.7 the simplest candidate for a collapsing procedure is demonstrated. Here, proving SQN-3 will ground SQN-2, which in turn will ground SQN-1. Therefore, the user might want to remove the clutter, created by all nodes of no interest between the chosen endpoints of the collapse (SQN-1 and SQN-3).

6.2.6.2 Multiple Subgoals Deduction Steps

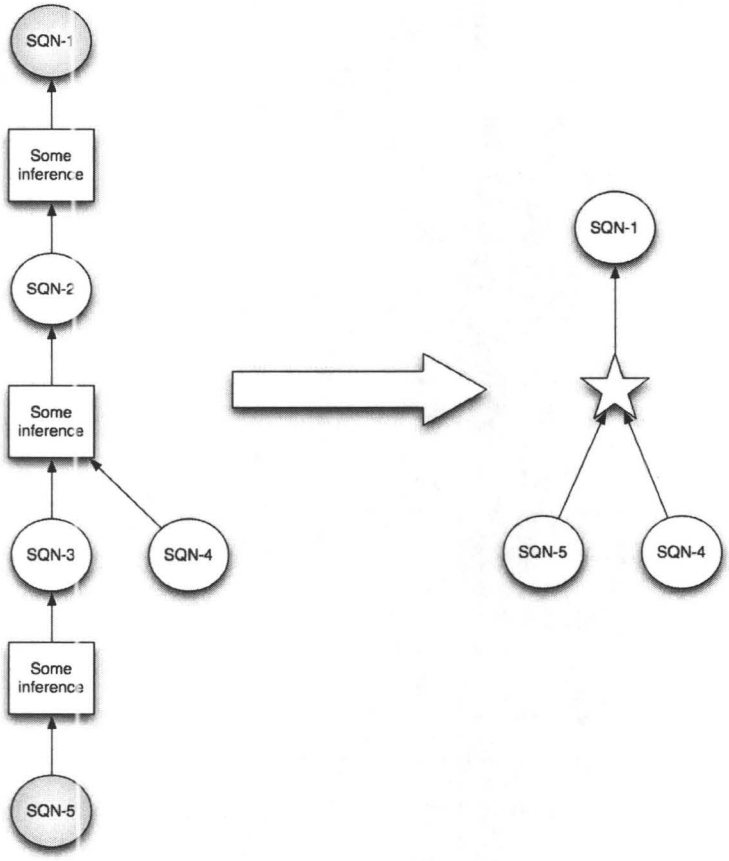

Figure 6.8: Collapsing parts with deductions with more than one subgoal.

When collapsing, a case might appear when one or more of the inference nodes between the chosen endpoints of the collapse procedure have more than one hypothesis (subgoal). Consider Figure 6.8: If we suppose that the user wishes to collapse everything between sequent nodes SQN-1 and SQN-5, then we can see that the procedure is not as straightforward as just replacing everything in between with a collapsed box. In fact, now we have a sequent node SQN-4, which is an extra subgoal introduced in the picture by some middle breakdown in the proof.

By definition, proving SQN-5 will ground SQN-3, which combined with a proof for SQN-4, will ground SQN-2, which in turn grounds SQN-1. Therefore, in order to ground SQN-1, we need to prove SQN-5 and SQN-4, which is graphically represented by the graph on the right in the figure.

6.2.6.3 Multiple Proof Directions (Branching)

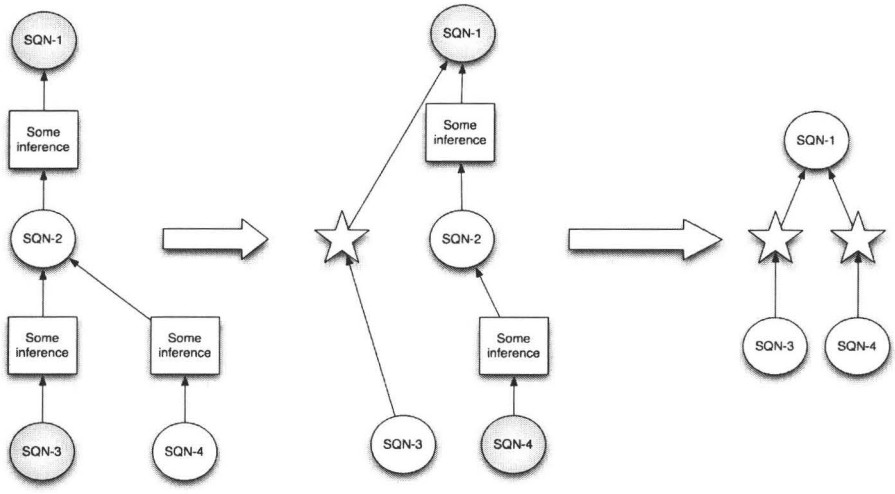

Figure 6.9: Collapsing parts with more than one proof direction

As we already know, sometimes the users might go back to the original goal or earlier subgoal, and attempt other procedures in their search for a proof. An example can be seen in the leftmost graph in Figure 6.9. Here, grounding either SQN-3 or SQN-4 will guarantee the upmost goal (SQN-1).

Consequently, the users might want to hide the information, which is not really helping in their search for a proof. Unlike the previous collapsing case scenario (Subsection 6.2.6.2), we cannot automatically hide everything between the leaves and the goal due to the fact that a star exhibits the same properties as an inference node. If we do that, the integrity of the graph will be destroyed. That is why the users should be able to decide which branch they wish to collapse, and the other branches should remain unchanged. Of course, the users may sequentially collapse all branches, as illustrated in the figure, where the end result will simply mean that SQN-1 can be grounded by either grounding SQN-3 or SQN-4.

6.2.6.4 Cycles

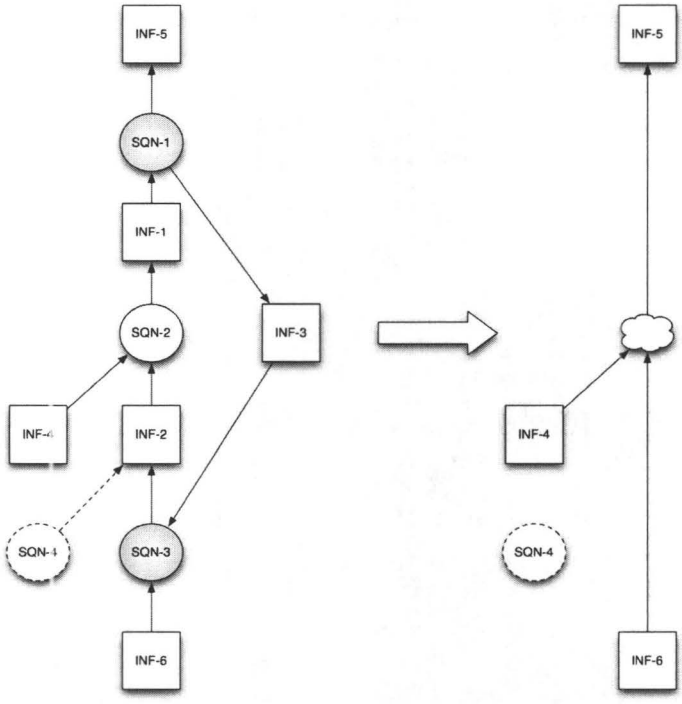

Figure 6.10: Collapsing cycles

Consider Figure 6.10[2], which shows a part of a deduction graph, containing a cycle. This example shows all possible situations in which a cycle can occur:

- INF-5 is an inference node, coming out of the cycle;
- INF-6 and INF-4 are inference nodes, coming into the cycle;
- SQN-4 is a sequent node, coming into the cycle (notice that there is no sequent node that comes out of the cycle, which is due to the fact that an inference may have one and only one child).

If the user decides to collapse (or hide) this cycle, they will have to select two sequent nodes in the cycle, and then invoke the cycle reduction functionality. Consequently, the program should verify whether these two nodes are a part of one common cycle. Unfortunately, the latter check is not enough—the program must also verify that

[2]The dotted line in the figure connects a node for illustrative purposes only. The presence of this node in the example will actually make it impossible to collapse the cycle, as it is explained in the text.

there is no sequent node coming into the cycle (in our example, such a node is SQN-4) because in this current arrangement it is not possible to replace the cycle with a cloud shaped node, exhibiting the properties of a sequent node. One possible solution to this problem will be to introduce a 'dummy' inference node, but this will be an invasive treatment of the graph, and it is not recommended. A much better option is to make the lack of such node a prerequisite for being able to collapse a cycle, which will not limit the user too much[3].

6.2.6.5 Converging Branches

Recall that sometimes the process of developing a proof will lead the user to go back to an earlier goal and attempt to proceed with different proof steps. It is possible that the proof leads the user to the very same sequent that was reached earlier by following a different approach. For the purpose of this document, we will use the term *converging branches* for such scenarios.

Consider the top left diagram on Figure 6.11. Clearly, the proof proceeded by reducing SQN-1 to SQN-3 by four different routes. If the user wants to collapse the part of the graph, which ultimately leads to the same sequent node, they would normally select the two ending sequent nodes as endpoints of the collapse (SQN-1 and SQN-3). After the collapse the diagram on the bottom should be obtained, which is clearly valid: SQN-1 reduces to SQN-3 by INF-5, and SQN-6, SQN-5, and SQN-7 become obsolete and can be removed from the graph. Unfortunately, the process is not so straightforward, unless there is at least one branch consisting of a direct reduction of SQN-1 to SQN-3 without any inference nodes with more than one child sequent node. Look at the top right diagram of the same figure, where no such option exists. In that case, collapsing everything will be wrong since there will be more obligations to be satisfied to preserve the meaning of the graph. Those are the combinations of SQN-3 and SQN-6, SQN-3 and SQN-8, SQN-3 and SQN-5, or SQN-3 and SQN-7. Therefore, it is apparent that Panoptes should reduce such parts of the graph if and only if there is one straight branch of reduction.

Another scenario is when there are two or more converging branches, but the user would like to get just some of the branches out of the way. Consider Figure 6.12, which

[3]Based on interviews conducted with users of IMPS, a cycle with incoming sequent nodes happens very rarely.

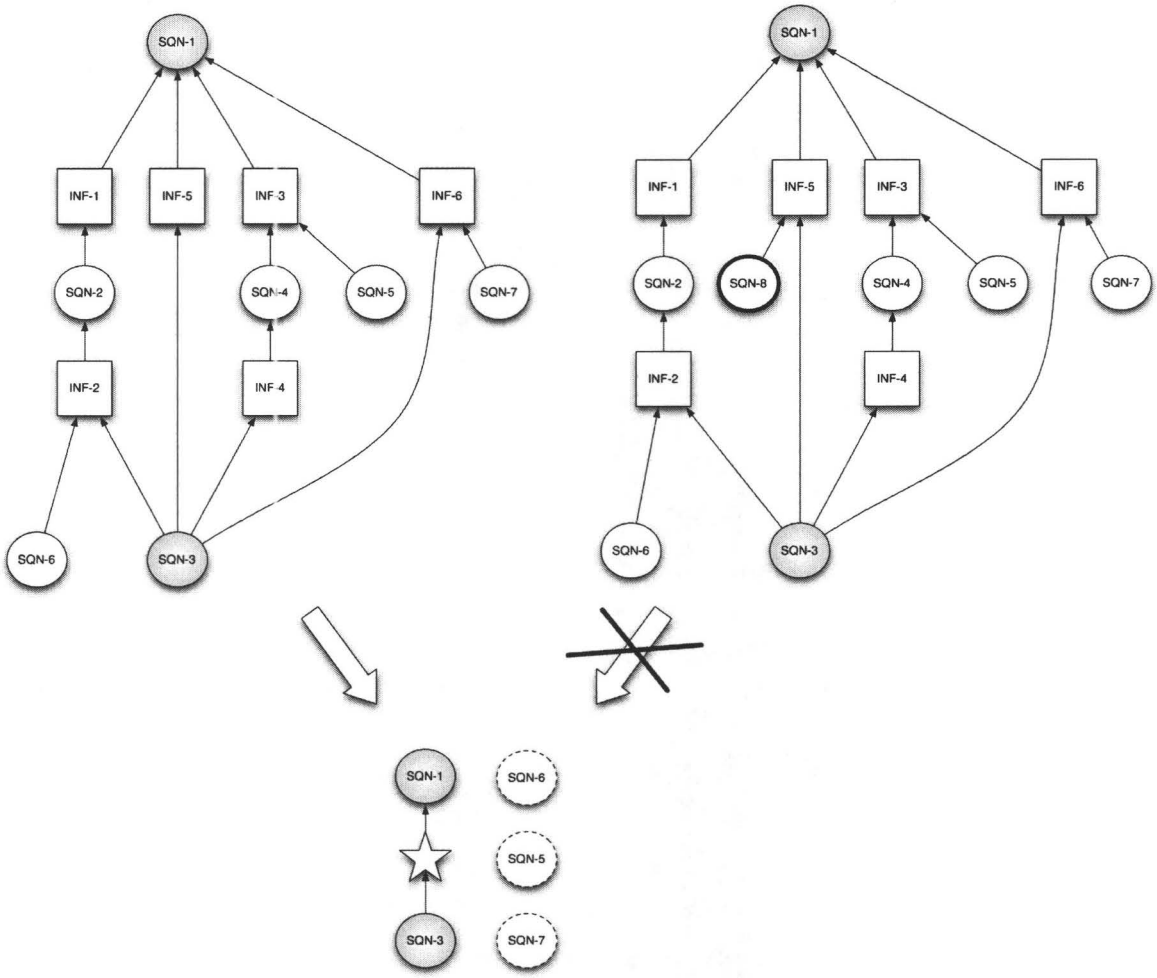

Figure 6.11: Collapsing converging branches

has two converging branches. The diagram on the right shows a state, in which the right branch is collapsed. This functionality should be available, but the selection method becomes more complicated. Now the user needs to have the means of making it clear which of all branches need collapsing. This can be achieved by starting the process as usual (selecting beginning and ending sequent nodes, between which the collapse should occur). Then Panoptes should scan the graph, and in the case that there is more than one possible branch to collapse, to prompt the user whether they want to collapse the whole part of the graph as described at the beginning of this section, or to select an additional inference node. Thus, having two sequent nodes and one inference node selected, Panoptes will know what the choice of the user is.

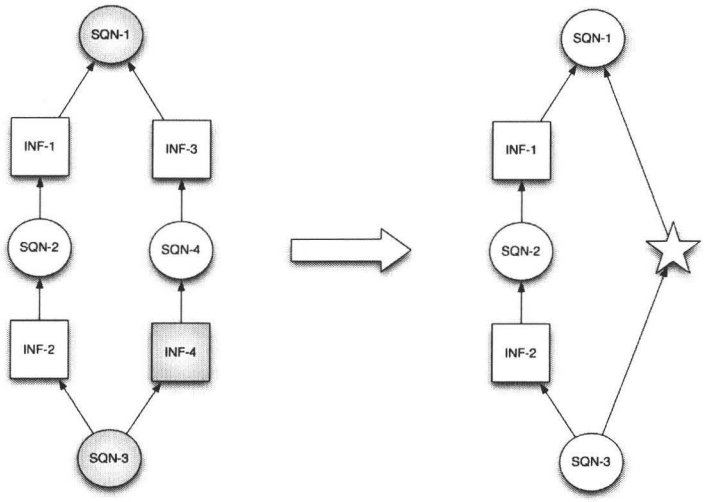

Figure 6.12: Collapsing just one of many converging branches

6.2.6.6 Examining Collapsed Nodes

One way of "looking into" the collapsed boxes will be to automatically produce an information box (described in Section 6.2.3) for each collapsed node (denoted by a star or cloud shape in our diagrams). Ideally, the information presented in these boxes should take the same form as the part of the graph, which was collapsed. There is a potential problem to this approach though: very often the collapsed part of the graph will be a long chain of deduction steps, each of which produces only one subgoal (see Subsection 6.2.6.1). Thus, the ratio between the height and width of the information box will be too large, and it will be just as unreadable on the screen as the part of the graph that was initially collapsed. Also, zooming in on the information box will not alleviate this problem any more than just zooming in on the graph itself before the collapse.

An alternative solution is to present the collapsed part of the graph in a way that IMPS currently presents the deduction graph in its user interface. For example, consider again Figure 6.11 from page 37. If the user wanted to examine the contents of the star node, which represents the collapsed part of the graph, they would see the following information in IMPS format:

```
(SQN-1
   (INF-1
      (SQN-2
```

```
            (INF-2
               (SQN-6)
               (SQN-3))))
      (INF-5
         (SQN-3 -see above-))
      (INF-3
         (SQN-4
            (INF-4
               (SQN-3 -see above-)))
         (SQN-5))
      (INF-6
         (SQN-3 -see above-)
         (SQN-7)))
```

Alternatively, the user can be given a further choice by the option to display just the direct branch, which is the basis of the collapse (in the case of converging branches). In that case, the information can consist of just the following:

```
(SQN-1
   (INF-5
      (SQN-3)))
```

A third, slightly different, but very useful and potentially best method will be the so called "momentary undo" function. It should allow the user to hold down a button while hovering over a collapsed node, and as long as the button is pressed, a temporary uncollapsing of the node should occur. If the performance of the program is good enough, the user should be able to switch back and forth until they comprehend the information they need.

6.2.6.7 Naming of Collapsed Nodes

The most meaningful way to name a collapsed node would be to construct its name from the beginning and ending nodes of the collapsed part of the graph. Consider again Figure 6.8. According to the suggested mechanism, the star-shaped node will be named "SQN-1 => SQN-5". Although this approach does not provide the complete information about the collapsed node, it certainly increases the information that can be apparent. In this example, it can be safely assumed that this node "*represents*

the part of the graph, which was collapsed after selecting SQN-1 and SQN-5 as end collapse points."

6.3 Integration with IMPS

This section analyses the way IMPS is designed, and creates a design for integrating Panoptes with IMPS so that they can work together.

6.3.1 How IMPS Works

Figure 6.13 shows how IMPS operates. The user interface is written in Emacs Lisp and runs in XEmacs or GNU Emacs, while the IMPS engine runs in Common Lisp in the background as a process, and accepts commands from the user interface. Once computations are done, IMPS sends back the results to the user interface, where further actions and commands are selected by the user.

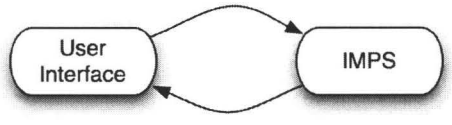

Figure 6.13: Communication between IMPS and its user interface.

Building the deduction graph can be viewed as a combined effort between IMPS and the user interface. When IMPS receives a sequent node and a command, it returns information about the newly produced nodes. The user interface then "plugs" this information into the right place in the deduction graph.

The deduction graph (DG) is stored on the disk in the form of a text file, which has the following format:

```
DG := --
    | SQN

SQN := (name INF_CHILDREN)

INF_CHILDREN := --
```

```
                | (name SQN_CHILDREN)

SQN_CHILDREN := SQN
                | SQN SQN_CHILDREN
```

When IMPS returns the result, the information about the newly produced sequent nodes is written on the disk. The user interface reads this file and appends the information to its own resident buffer in Emacs.

6.3.2 Actions, Induced by IMPS and Its User Interface

Typically, the user invokes commands from the user interface on chosen sequent nodes. IMPS receives these "requests," does computations, and returns a result. Panoptes should be 'aware' of such events by eavesdropping on the communication protocol between IMPS and the user interface, and more particularly, on the return of the results from IMPS to the user interface. This protocol basically consists of modifying files on the disk, so Panoptes should implement constant monitoring of these files. Consequently, Panoptes should notify the user when changes to the deduction graph are detected.

6.3.3 Actions, Induced by the User

The requirement for Panoptes to notify the user upon changes in the deduction graph was described in the previous subsection (6.3.2). When this happens, the user should have the choice to either continue working on the deduction graph as it is, or to introduce the new information into "the picture." This process should be straightforward and painless, possibly with the hit of a button. Then, Panoptes will redraw the whole deduction graph along with all collapsed sections in it, and allow the user to continue their exploration and work.

6.3.4 Integrating Panoptes In the Process

Since the deduction graphs are always present on the disk in their complete form, Panoptes needs to read that file, parse it, and create a graph in its own data structures.

Additionally, similar to the user interface, Panoptes also needs to keep track of new information in the file that holds the info for the sequent nodes, and whenever there is something new, it should collect it and store it. This is needed because with each step, the file gets overwritten with the information about the newly produced nodes, which is the way IMPS was designed and functions.

6.3.4.1 Flow of Commands (Messages)

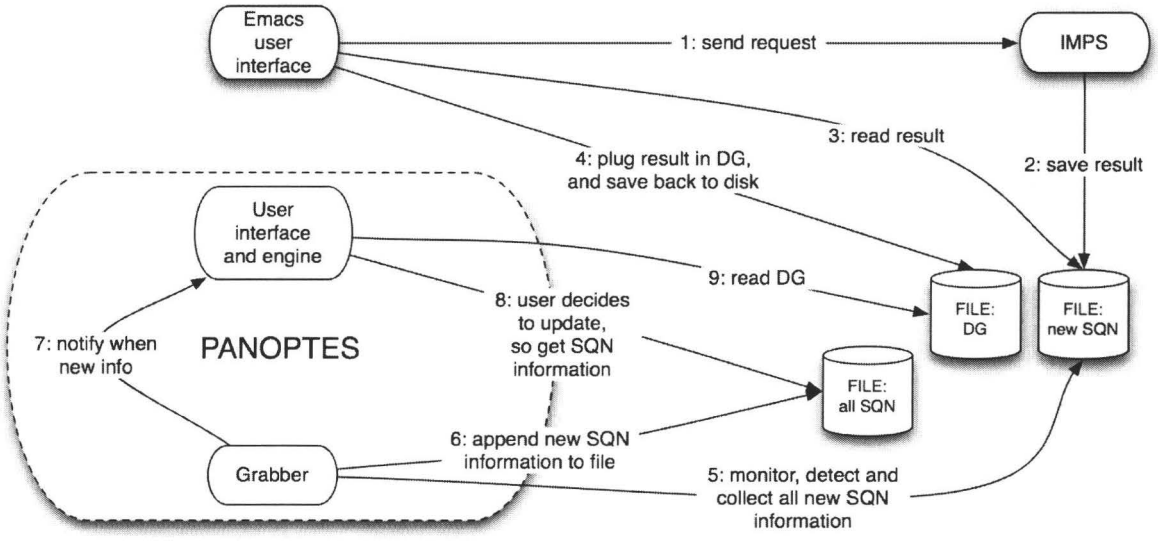

Figure 6.14: Exchange of commands in the system

Consider Figure 6.14. It shows how IMPS, its user interface, and Panoptes combine to form one big system. In the figure, the edges are numbered and labeled. If the whole work process was divided in smaller sequential steps, the numbers would represent the order, in which the events occur. The process starts with the user choosing a goal in the Emacs user interface, which comes with IMPS. Then, the user decides to apply a certain command to this goal, so this information (the goal and the command) are sent to IMPS (edge 1). When IMPS finishes its computations, it saves the newly produced sequent nodes on the disk (edge 2). The Emacs user interface then reads that information (edge 3), stores it in a resident buffer of its own, plugs the new nodes into the deduction graph (DG), which is then written back to the disk by overwriting the file that stores it (edge 4). Meanwhile, the Grabber module of Panoptes detects availability of new sequent nodes information, so it collects it (edge 5) and appends it to its own file on the disk, which is exclusively for use by Panoptes (edge 6). Then

the Grabber sends a message to the user interface (edge 7), which makes the user aware that there is an available update to the deduction graph. At this point, the user might decide to continue working without updating the graph, during which time steps 1 through 7 might repeat without loss of any information. When the user decides to update, Panoptes reads and updates its internal data structure with the new information about sequent nodes (edge 8), and then reads the final deduction graph (edge 9) and rebuilds it internally. Then, the user is presented with the new deduction graph visualization and he or she may continue working.

6.3.4.2 Data Flow

In reference to Figure 6.15, unlike the previous diagram, which showed the commands sent from components of the system to other components, the diagram in this figure describes the data flow in the system. The dash-point-dash lines connect any two components that may exchange some kind of data at a certain moment during runtime. Also, these lines are directed and labeled with a short description of the data that is being exchanged. An arrow with a label "abc" from a component A to a component B indicates that the component A sends the data unit "abc" to the component B.

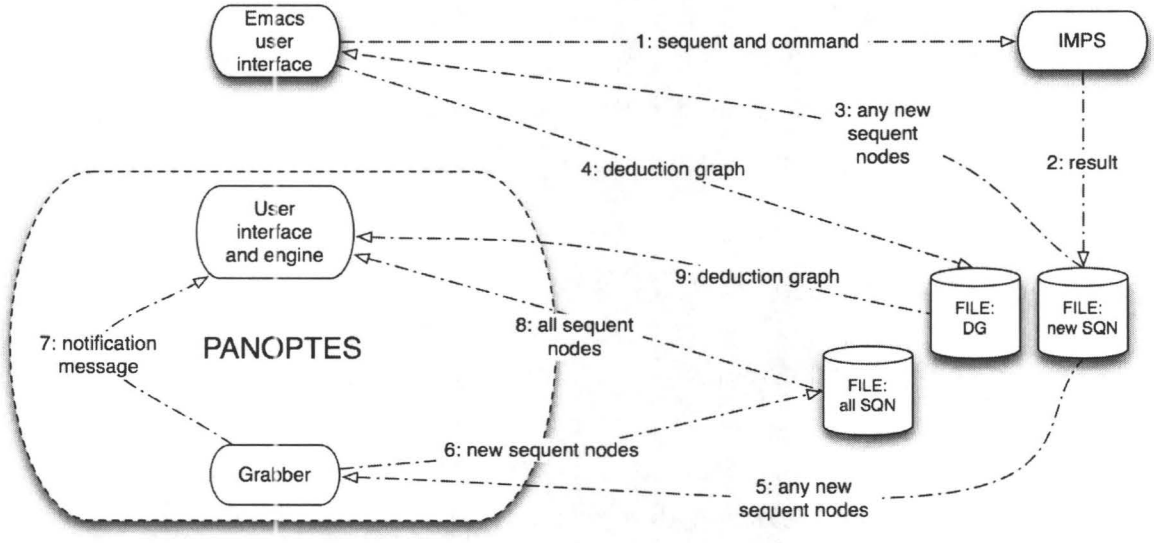

Figure 6.15: Data flow in the system

The reader will also notice that the numbering from the diagram in the previous figure is preserved to ease the understanding of the process. This is possible because of the fact that the data exchange happens in the same sequence as the one for messages exchange, which was described in the previous subsection (6.3.4.1). Rather than being an accident, this happens because of the sequential manner in which proof development events happen in the system as a whole.

6.3.5 History of Operations Between Deduction Steps

Suppose the following situation is at hand: The user works on the deduction graph in Panoptes by rearranging parts of the graph, collapsing parts of the graph, renaming nodes, opening and situating node information boxes. At that point, the user has customized the appearance in a way, suitable to visually inspect and make judgments. Then, the user decides to work on a goal, and selects a suitable command from the user interface, which in turn passes it to IMPS, and the latter produces a result. At that point, the graph has new additions to it, and the user is notified (See Subsection 6.3.2). Consequently, the user chooses to update the graph in Panoptes by invoking the update functionality (See Subsection 6.3.3). At that point, if precautions are not taken, Panoptes updates the graph and redraws it. As a result, all rearrangements to the graph, mentioned above, will be lost. That is why, Panoptes needs to keep track of all actions by the user, and when the graph is updated, to automatically execute them. It is important to note that throughout the life cycle of the search for a proof, no information ever gets lost, that is, only new information is being introduced into the graph. This implies that most of the actions on the graph will always be valid, except the case described in Section 6.2.6.5. Panoptes should automatically detect this case with converging branches and disregard the command.

6.4 Structuring the System

The design of the system follows an object-oriented approach, where each module constitutes one class. These classes can be instantiated into objects by the objects of other classes in the system to provide services to them. Furthermore, each module is separated in a single file on the disk, which must be included by the modules that need to use it. Thus, an easy identification of the relevant part for each particular service can be identified and examined easily. Refer to Appendix A for a detailed

description on how Panoptes is modularized, as well as a detailed description and dependencies between modules.

6.5 Shortcomings of the Design

A design created concurrently with the process of gathering requirements runs the risk of not being the best possible design. However, the main goal of this thesis is to suggest a system that solves the problems associated with the limitations of the current user interfaces of theorem provers. The current design and implementation (see Chapter 7) offer a working prototype of Panoptes, but should not be considered as a final product to be released and used for heavy and possibly commercial work.

That being said, there is an ample room for further analysis and refinements. For example, the design will benefit from a description in the form of an industrially established method, such as UML (class diagrams, object diagrams, collaboration diagrams, etc.). Furthermore, the design can be formalized and then verified for correctness by using software tools, such as the B-method, or even IMPS. However, that process is lengthy and cannot be completed in the available time frame.

CHAPTER 7

PROTOTYPE IMPLEMENTATION OF THE SYSTEM

7.1 Overview

This chapter describes the implementation of a prototype of Panoptes. There is a description of the chosen programming language and libraries that are used for different tasks. Also, the end of the chapter provides detailed instructions on how to install the software.

If the reader desires to gain a deeper understanding of the implementation, he or she should refer to Appendix B, which contains comprehensive documentation of the source code. For the ultimate enthusiasts and also developers who wish to extend the code or use parts of it in their own programming, the fully commented source code can be downloaded and viewed from the link provided at the end of this chapter.

7.2 Programming Choices

One of the most important nonfunctional requirements for Panoptes is to draw graphics fast enough to eliminate any noticeable lags for the user. This is not only dependent on the design of the system, but also on the implementation methods, which in turn requires a good choice of a programming language and libraries for graphical

support. This section describes the choices that were made in this regard.

7.2.1 Objective Caml (vs. C/C++)

The importance of the choice of a programming language comes from the fact that the compiler bears the task of transforming a high level program into low level executable code that runs sufficiently fast on the host machine. This requirement is vital for avoiding the situation when the user has to wait for the program to finish its calculations. The latter could not only result in wasting the user's time, but might also break his or her focus and concentration on the problem. Some might argue against the importance of this requirement, but experience has shown that even a little slowdown might be detrimental to the user having a pleasant experience. Recall that one of the main goals of Panoptes is to make computer assisted theorem proving a helpful and accessible aid to the user rather than create one more reason to continue producing proofs in the old-fashioned way with paper and pencil.

Objective Caml [13] (a.k.a. OCaml) is a programming language created by Xavier Leroy, Jerome Vouillon, Damien Doligez, Didier Remy, and others in 1996. Since then, OCaml has established itself as a popular open source ML-derived language that can be used not only for specific tasks but also for generic programming. It is currently managed and maintained by the French National Institute for Research in Computer Science and Control (INRIA) [12] and is characterized by a very large community of users both in Europe and North America.

One of the most prominent characteristics of OCaml, which was also the main reason to choose it for implementing the Panoptes prototype, is the static type system that is also complemented by a powerful type inferencing engine. This feature alone excluded the usual choice, C or C++ for object-oriented programming, made by most programmers for implementing such projects. This type inferencing capability of OCaml provides an unmatched aid in producing code free of programming errors. On the occasion of some type mismatch in the program, which is one of the easiest errors to make but among the most difficult to discover, the compiler will automatically output an error. In contrast, the C++ compiler will produce a runnable program that will ultimately lead to, for instance, a "segmentation fault" during some point of a runtime session. Thus, OCaml guarantees to a large degree that a program that compiles successfully has a much lower chance of crashing than a similar program

written in C/C++.

Furthermore, the OCaml programmer does not need to worry about garbage collection as the latter is done automatically by the automatic memory management and incremental garbage collection system built into the language. This means that incidental, but painful, omissions of the "new", "malloc", "delete" and "free" operators in the C family of languages are not a problem anymore (in fact, OCaml does not even have such operators). In addition, OCaml offers the programmer an environment for not only using imperative programming and object-oriented constructs, but also functional programming, which is a capability directly derived from the OCaml predecessor, ML.

Most importantly, programmers in OCaml enjoy the countless libraries available for dealing with almost anything from pattern matching (which is, by the way, built-in to OCaml) to creating large-scale graphical applications. In addition, studies show that the performance of a program compiled with OCaml is comparable to the equivalent code written in C++ [19], and the compilation process is just as fast, and in some cases, even faster than the similar process in C++. Furthermore, a practical comparison of a Ray Tracer implementation in both C++ and OCaml [3] reveals that OCaml code is about twice as short and succinct as the equivalent code written in C++.

Last but not least, a large amount of literature on developing a full range of software in OCaml is available in the form of online guides and tutorials, as well as reference books and textbooks. This was essential for developing the Panoptes prototype since the author's experience in developing OCaml programs was initially limited to a short programming assignment, which only touched upon the world of OCaml.

7.2.2 OpenGL

The performance level of Panoptes is not supposed to differ much from that of a fast paced video game. This type of applications require immediate display of the relevant graphics since even a little delay might make the display of a certain graphical component obsolete. To require such graphical performance from Panoptes means that the implementation of the project is more complicated than the implementation of applications that just draw graphics on the screen without interaction from the

user. Thus, the dynamic interaction between the user and the components of the visualized deduction graph implies that Panoptes should respond to each action in a quick, almost immediate, manner.

Panoptes uses the LablGL library [8] that implements an interface to OpenGL [10] in OCaml. Usually, OpenGL is associated with three-dimensional graphical visualizations but the prototype uses these capabilities for implementing different techniques, some of which are described below.

7.2.2.1 Three-dimensional Space and Zooming

The deduction graph in Panoptes is an object that resides somewhere in the realm of virtual space. The computer screen can be viewed as a window to the space, from which the user can peek. Therefore, only a small section of the whole space is visible by the user at any given time. This section is defined by the position coordinates of the user (resp. the computer screen) as well as the direction the window is pointing at (i.e. a vector). Also, further configuration is needed: the length of the viewing vector (the user cannot see things that are too distant), the field of view (a.k.a. FOV, measured in angle radians), etc. These features are supported by the 3D functionality provided by OpenGL, and Panoptes utilizes them for implementing some visual tricks, rather than using the API to draw graphics that are intended to represent objects that appear 3D to the user.

One such trick is used in the implementation of the zooming function. Moving the graph or selected components of the graph closer or further away from the viewer creates the effect of zooming, which mimics how zooming happens in nature—the closer an object gets to the eyes of the user, the bigger it appears and more detail is visible.[1] In contrast, the usual zooming method that is used in most applications is to merely scale the image. The advantage of bringing the graph in the space lies in OpenGL being a direct API to the 3D instruction set of the graphical hardware, and as such, it provides a performance unmatched by the standard way of drawing graphics on the screen. The result is an application which delegates all graphical computations and manipulations to the GPU, rather than the CPU of the host machine, thus leaving the latter fully available for other work (such as that done by the IMPS

[1] Alternatively, the user (represented by a viewport in OpenGL) can move closer or further away from the object to achieve zooming. Performance wise both methods are equivalent.

reasoning engine). Consequently, a computer system equipped with a reasonably modern graphics card would be capable of running the prototype smoothly without burdening the user with unnecessary lags and delays.

7.2.2.2 User Interface Elements

OpenGL comes with GLUT (The OpenGL Utility Toolkit) [26], which is a window system independent toolkit for writing OpenGL programs. It represents a library of utilities for performing system-level I/O with the operating system running the OpenGL program. The implementation of Panoptes uses GLUT for many functions such as window initialization, window control, as well as attaching event listeners for all keyboard and mouse activities.

The main reason for choosing GLUT for building the Panoptes user interface was not its full set of functions but rather its portability. GLUT makes it possible for the developer to not worry about the kind of host operating system that will be running the application. This is an important feature that perfectly fits the portability nonfunctional requirement for Panoptes.

7.2.2.3 Text in OpenGL

The Panoptes requirements to label all nodes in the deduction graph and to display information boxes with detailed information about the nodes calls for choosing a technology for displaying text in an OpenGL drawing area. This proved to be one of the most difficult issues to deal with during the process of developing the prototype because OpenGL has very little support for displaying text.

At the time of developing the Panoptes prototype, there were just a few known methods for displaying text in OpenGL programs [9]. One way of achieving this is to use bitmap fonts, which are usually very fast to display since they are pre-rasterized. Despite the speed advantage, though, the provided OpenGL functionality does not support rotation and scaling of the text. To make things even more limited, the method relies completely on the available bitmap fonts installed on the host machine.[2] However, the prototype implementation uses this method to display a help screen with available commands to the user upon request.

[2]Bitmap fonts are not the same as true-type fonts (TTF), which are vector based.

Another method for drawing text in OpenGL is to use external libraries for generating outline fonts. This approach converts each symbol of the alphabet into a polygon that can later be used in OpenGL as a separate 3D object. It is obvious that this method would be an excellent choice if the top priority is to display text with highest possible quality. Furthermore, these polygons can be manipulated in every possible way supported by the general OpenGL functionality (rotation, translation, scaling, coloring, lighting, etc.). Unfortunately, having a polygon for each character in a certain fragment of text would require OpenGL to render each symbol every single time a new frame is drawn to the screen. That is why this method was not utilized by the Panoptes implementation as it would have had detrimental effects on the smoothness of the animation.

The third method for displaying text in OpenGL is to represent the characters of the alphabet as textures. OpenGL provides extensive support and optimization for dealing with such textures, allowing them to be used as surface decals regardless of the size and location of the surface. In the case of Panoptes, the set of all individual surface units that are candidates for such decals includes all nodes and all information boxes. Consequently, this method was modified to convert True-Type Fonts into extremely high resolution images. Then an image containing the complete set of symbols is parsed to separate each symbol from the others, and Panoptes loads the whole alphabet in the form of generated OpenGL textures into the video memory. When the program needs to display text on the screen it just sends an OpenGL command to the video card with instructions on the textures to be used and the format and display location. This method proved to be extremely reliable and efficient because it completely bypasses the need for using the CPU of the system.

7.2.2.4 OpenGL Literature

Learning to do graphics with OpenGL has a steep learning curve. However, there is a tremendous amount of literature available in the form of references (e.g. [27]), textbooks (e.g. [29]), as well as online tutorials and guides. Most of these texts include developing sample projects that guide the user step by step into mastering techniques that gradually increase in difficulty. It is important to understand that OpenGL is not a programming language—it is an API to the graphical hardware, so it is the user's responsibility to understand the possibilities as well as the limitations

of OpenGL when it comes to drawing graphics.

7.2.3 Graphviz

Graphviz [24] is a software package developed by AT&T Research for creating layouts for graphs. Currently the software is licensed on an open source basis under the Common Public License stated on the following web page: `http://www.graphviz.org/License.php`.

Panoptes uses Graphviz for creating a layout of the deduction graph. The prototype describes all nodes and connections between them using a simple file format called *dot*. Then Panoptes runs this file through Graphviz, which in turn renders a layout by using sophisticated layout routines. The output is then parsed by Panoptes into an internal data structure, which is made available to the other parts of the system.

Since the performance requirement is crucial for Panoptes, a few different tests were created and executed to determine if Graphviz could hold up to the expectations. The results showed that Graphviz completes generation of layouts for deduction graphs containing a couple of thousand nodes in a fraction of a second, thus making it a great choice for use by Panoptes.

7.3 Installation and Use

The source code of this implementation of Panoptes can be downloaded from the following link:

`http://imps.mcmaster.ca/ogrigorov/panoptes/panoptes_src.tar.gz`

The file should be saved to a local folder (e.g. `/home/username/panoptes`), and then uncompressed by using the following command: `tar zxf panoptes_src.tar.gz`.

The user should run `./install`, which will create a new subfolder, containing high resolution images for each character of the alphabet. Alternatively, if the user has the package Camlimages installed on his or her machine, executing `make generator` and then `./generator` will generate the required character image files.

Before successfully compiling Panoptes, there are a few things that must be installed on the system. These are:

- **OCaml:** compiler for Objective Caml programming language.
- **LablGL:** OCaml implementation of OpenGL API.
- **Graphviz:** layout generation program.

If compiling on a Debian-based Linux, such as Ubuntu Linux, the process can be completed with a straight forward one line command: `sudo apt-get install ocaml lablgl graphviz`.

Mac OS X users need to install both Fink (`http://www.finkproject.org/`) and MacPorts (`http://www.macports.org/`), and after making sure that they are both up to date, to install the required programs by typing the following two commands: `sudo fink install ocaml lablgl`, and then `sudo port install graphviz`.

Once all prerequisites are present, Panoptes can be compiled by just typing `make`, which will create a binary file. Then, the system can be started by `./panoptes`, and the help screen with all available shortcuts can be called by pressing the "h" button on the keyboard.

Note that it is vital to start a proof in IMPS[3] before starting Panoptes. If this is not done, Panoptes will look for the IMPS communication files and will not find them, which will result in an error message displayed on the screen.

7.4 Demonstration

This prototype is graphical software with user interaction, and as such, there is no better way of describing it than demonstrating it live. That is why the home web-page of Panoptes (`http://imps.mcmaster.ca/ogrigorov/panoptes`) contains a demo that can be downloaded and viewed with almost any contemporary software media player.

[3]IMPS and information on how to install it can be obtained from `http://imps.mcmaster.ca`.

7.5 Screenshots

For completeness, this section provides a few screenshots of the Panoptes prototype. Such static images only begin to demonstrate the ease of operation and the features the program provides (such as the effective "target zooming"), so the reader is advised to visit the webpage mentioned in the previous section to see a live demo.

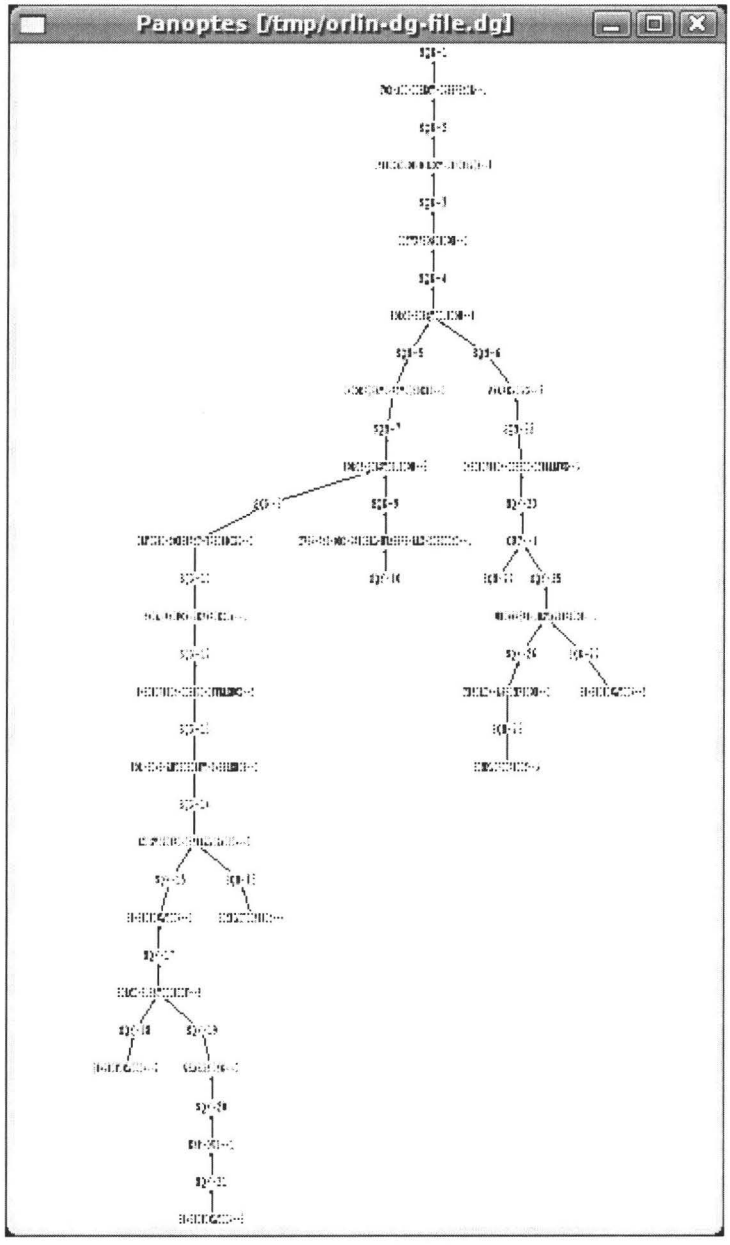

Figure 7.1: Screenshot: a deduction graph

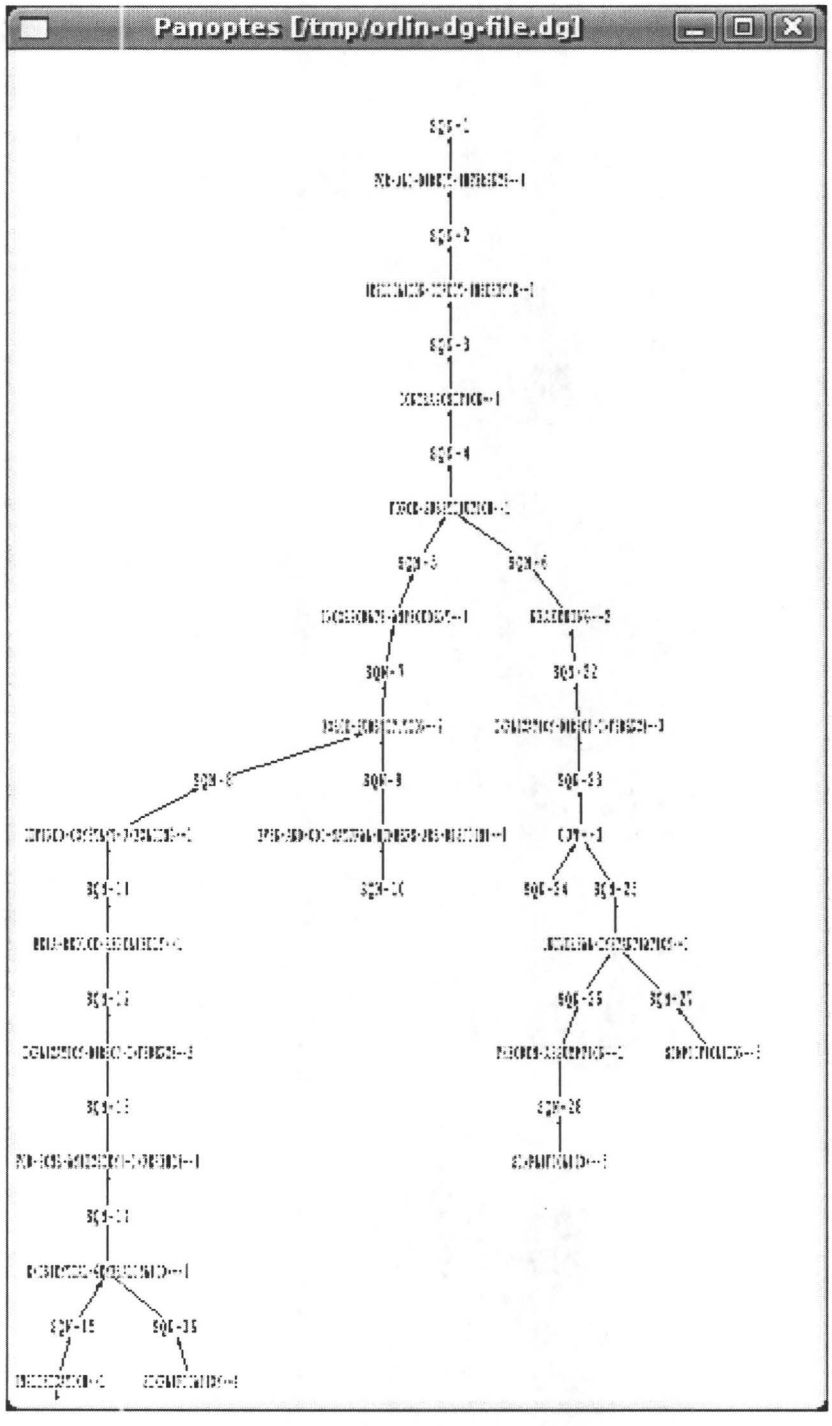

Figure 7.2: Screenshot: slightly zoomed and repositioned

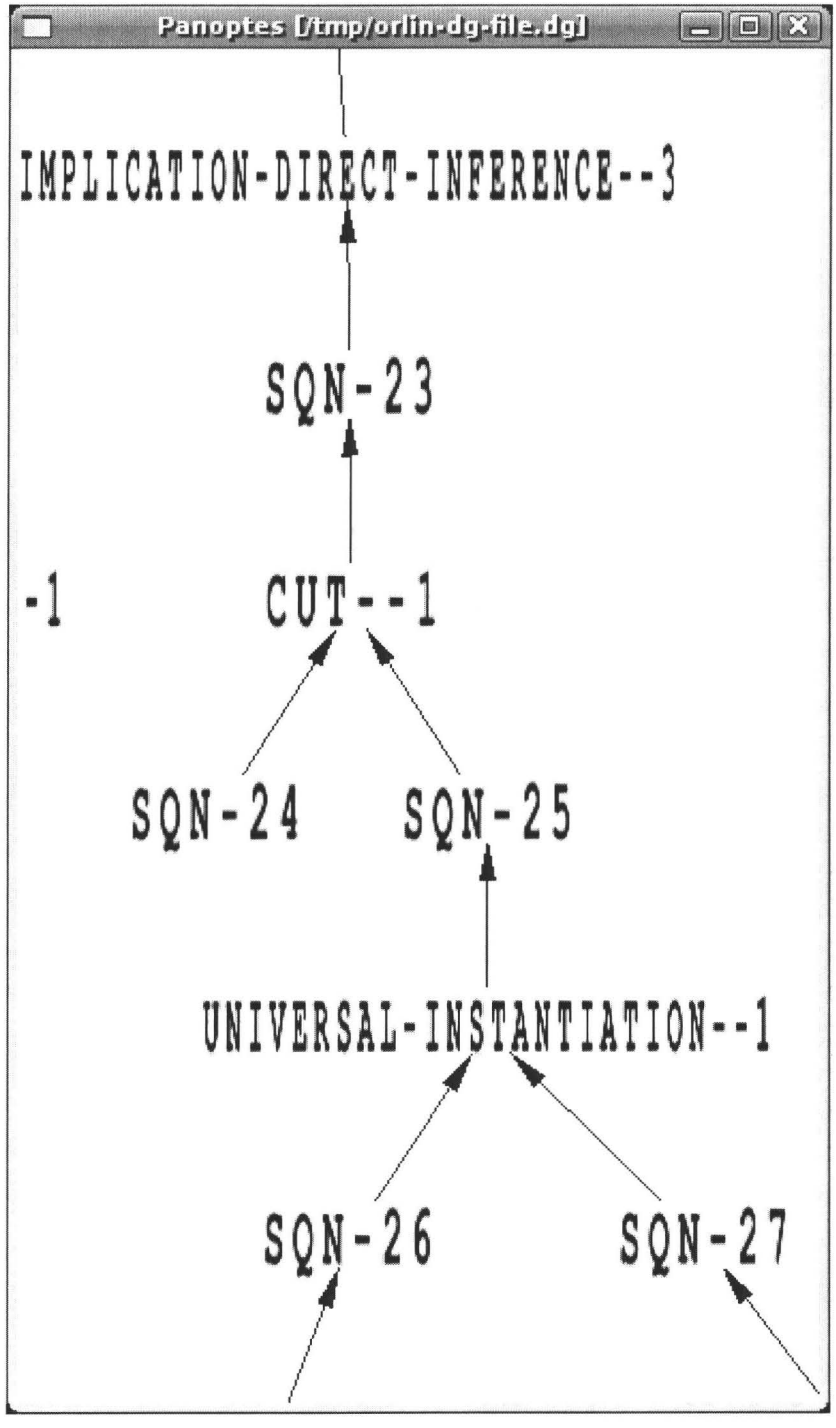

Figure 7.3: Screenshot: target zoom on "SQN-25"

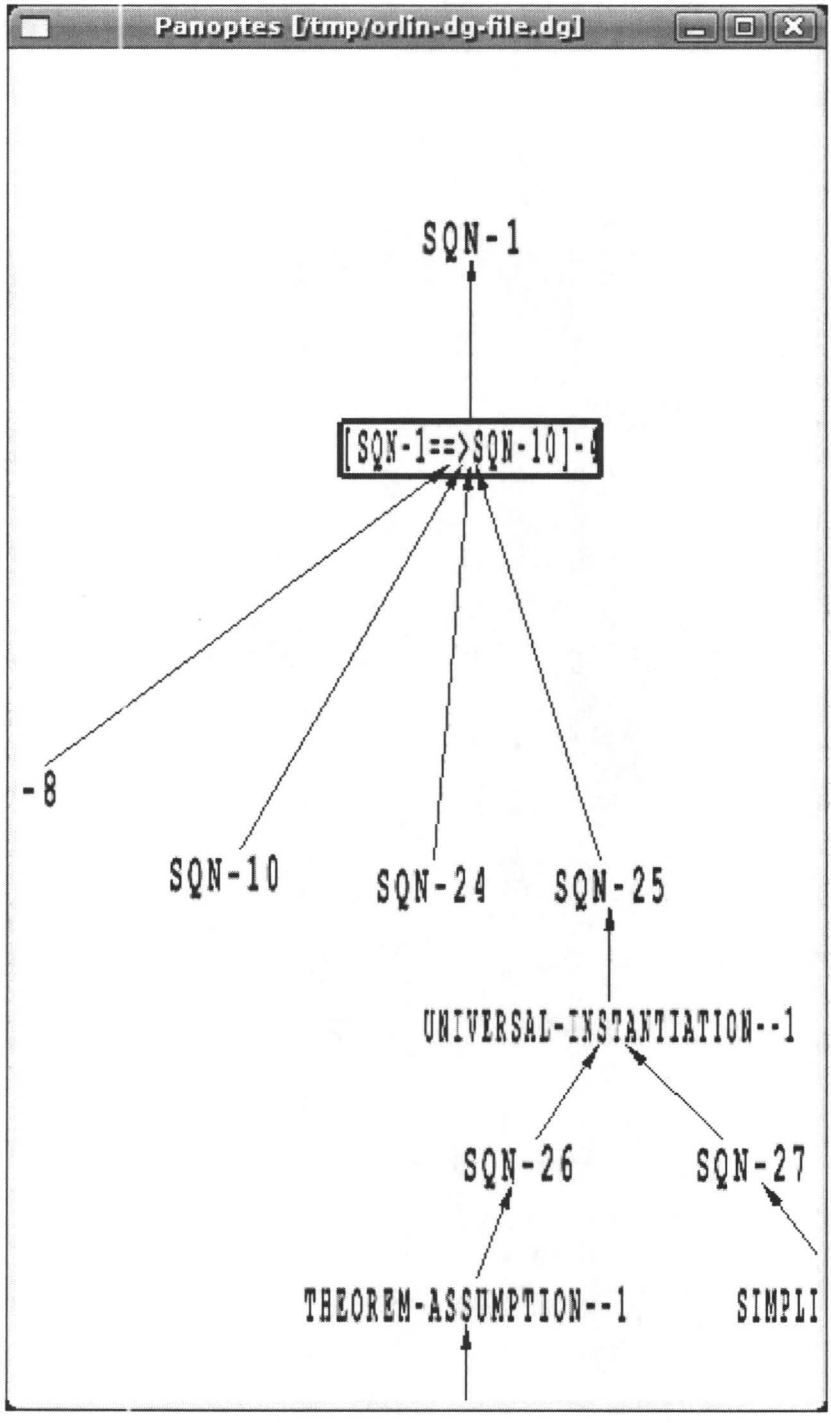

Figure 7.4: Screenshot: partially collapsed

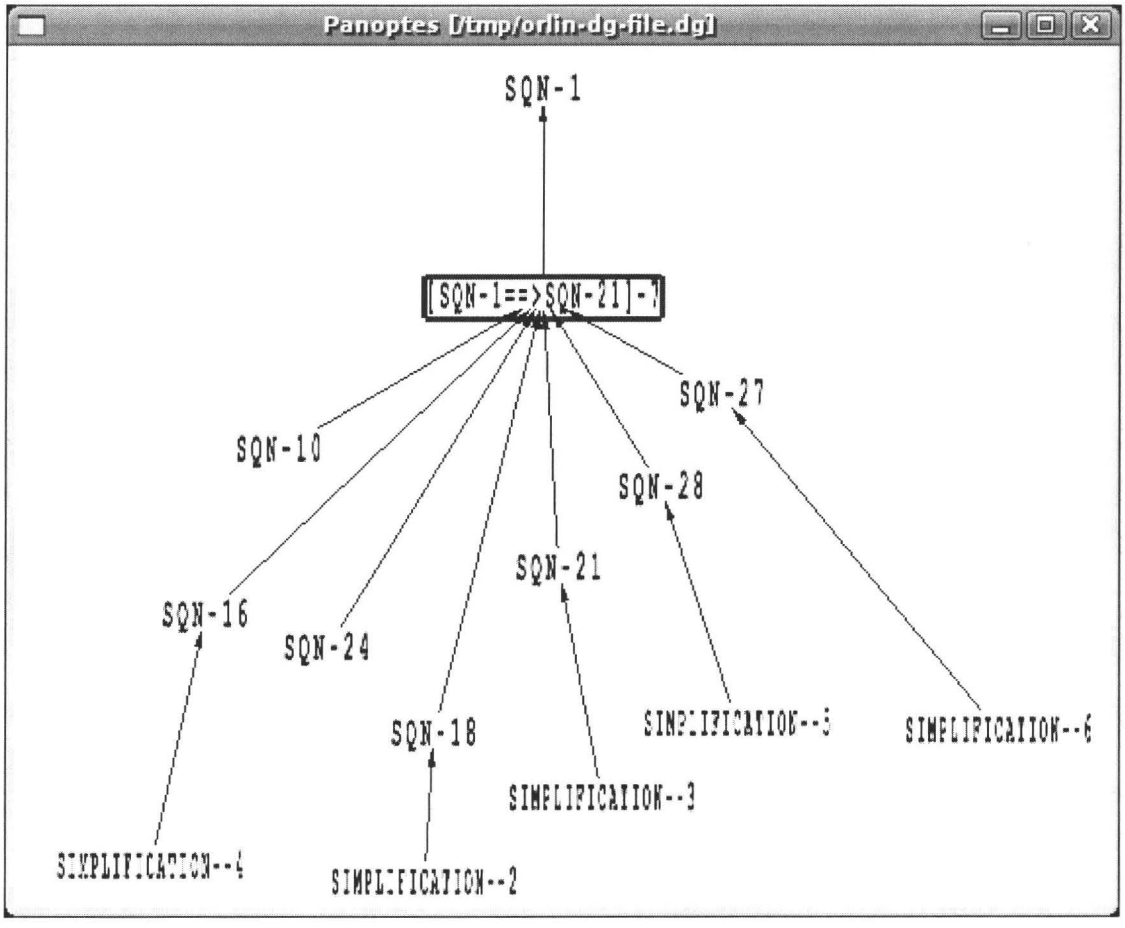

Figure 7.5: Screenshot: completely collapsed

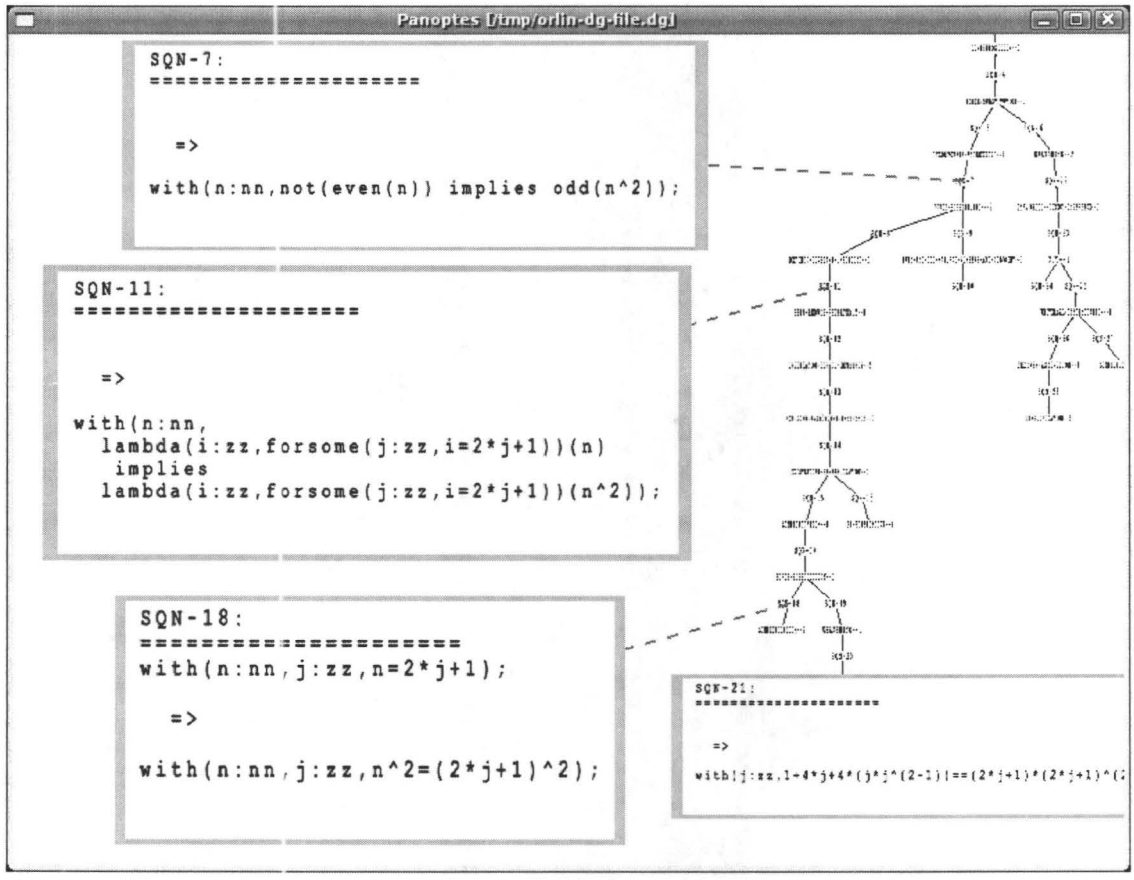

Figure 7.6: Screenshot: a few content boxes are opened

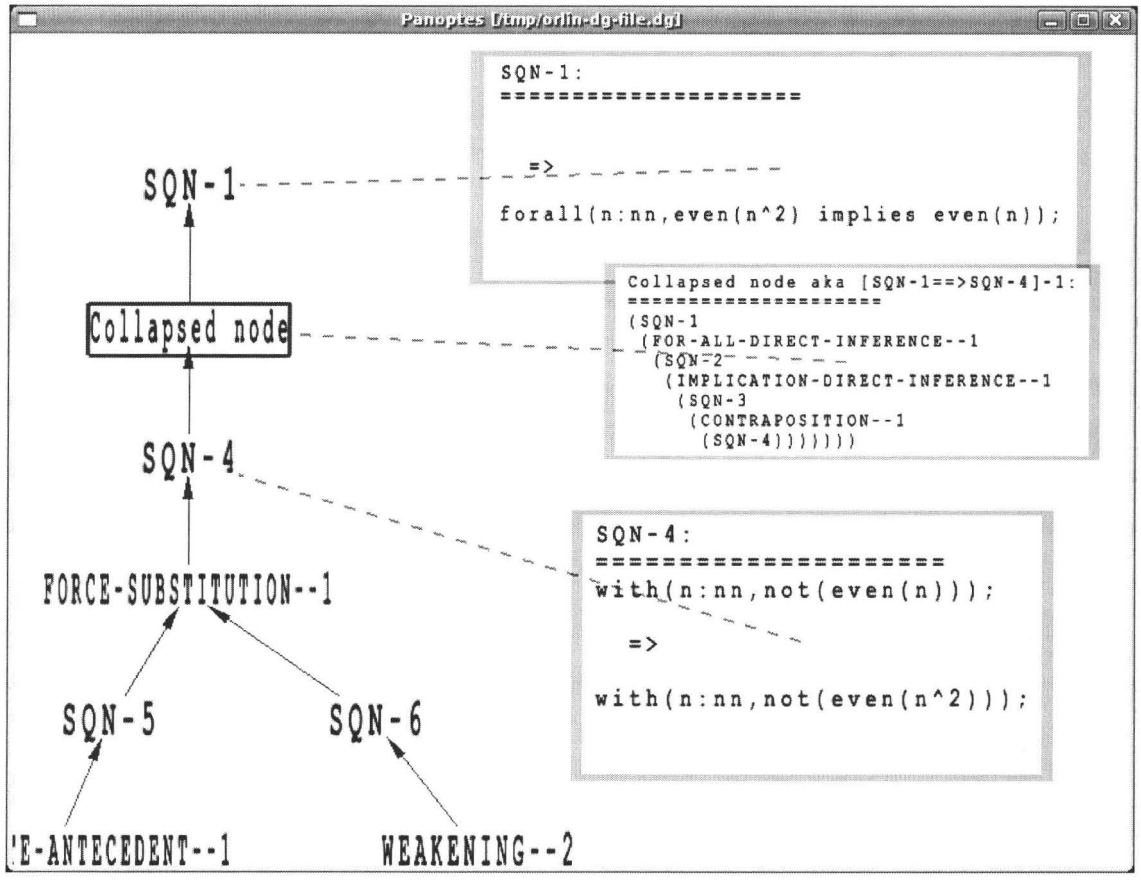

Figure 7.7: Screenshot: content boxes and a renamed collapsed node

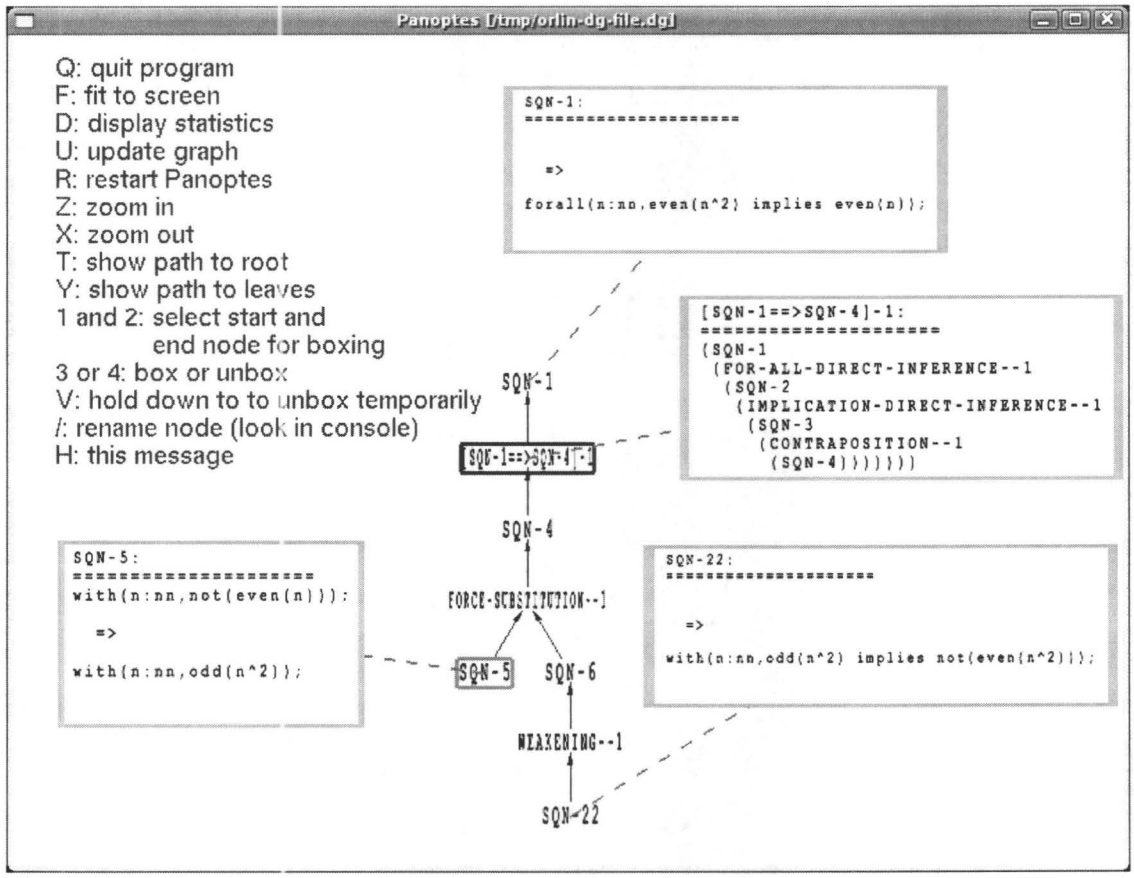

Figure 7.8: Screenshot: help screen, content boxes, grounded node "SQN-5" colored in green

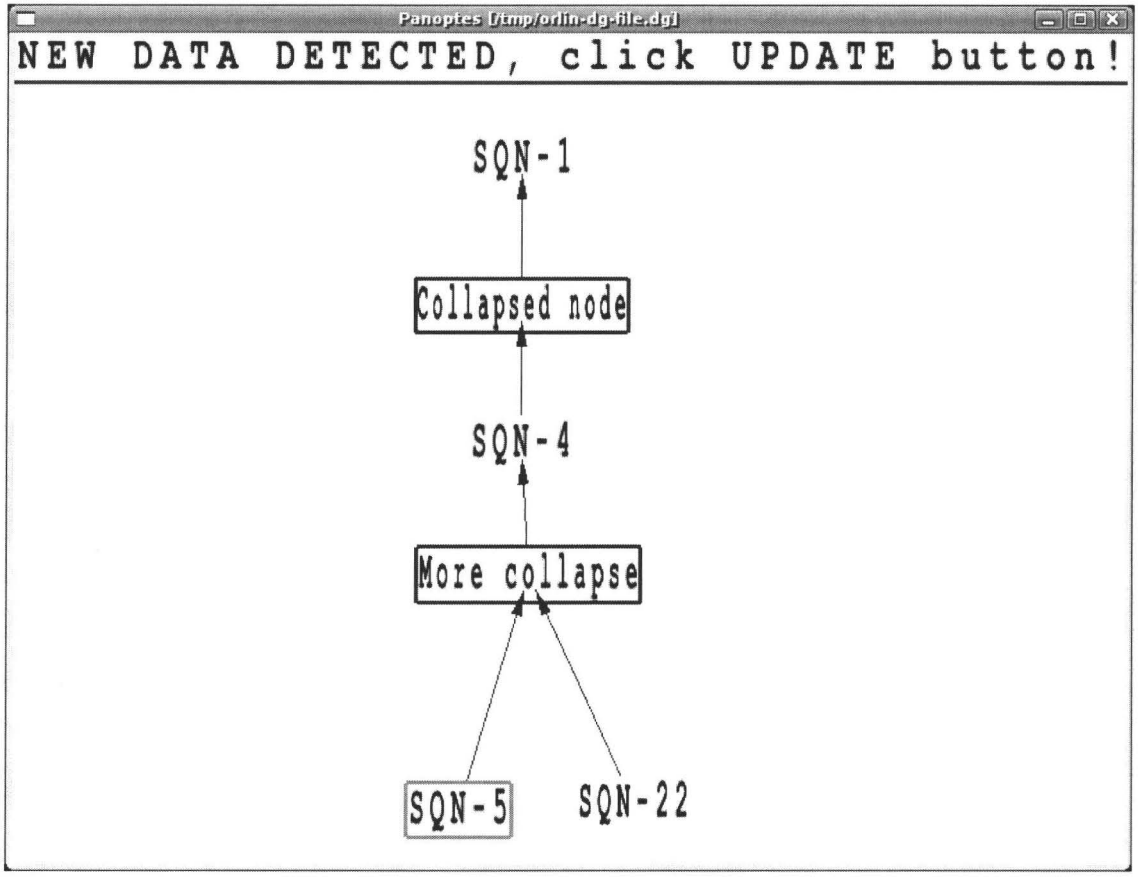

Figure 7.9: Screenshot: new data notification screen, renamed nodes

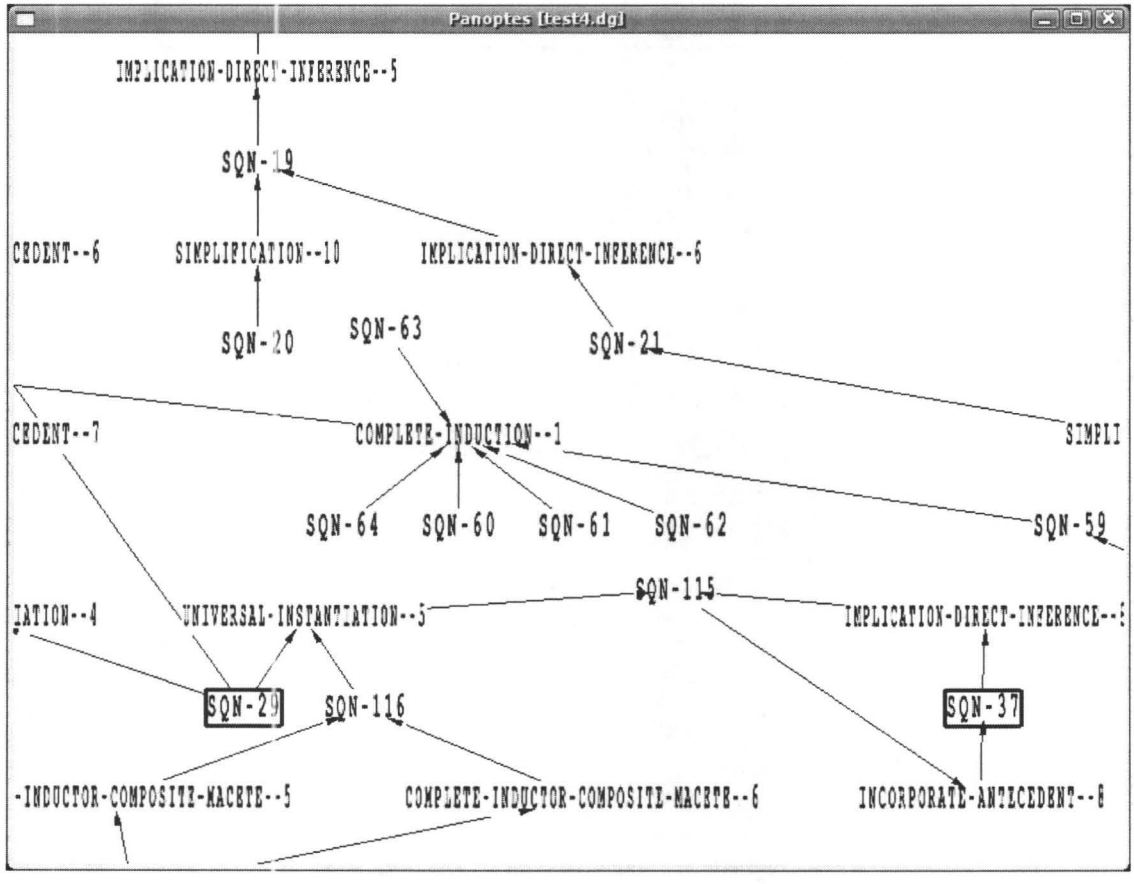

Figure 7.10: Screenshot: a zoomed section of a very large deduction graph

CHAPTER 8

RELATED WORK

The authors of [1] have developed a proof visualization system, which attempts to make proofs created with the ACL2 theorem prover [15, 16, 17] more understandable. As in IMPS, ACL2 relies on large amounts of text to represent theorems and steps in the proof process. The developed system has different design and requirements from Panoptes, though. First, its layout mechanism relies on a variant of the cone tree layout algorithm, which renders the nodes in 3D space. Consequently, the user is given the functionality to rotate the whole structure and look at it at all angles. For making visual differentiation between the nodes, they use colors instead of labels, where each proof command has its own color. Like Panoptes, they have implemented a way for examining the contents of each node, although this happens in a separate window and there is no visual connection between the node and its content box, thus making it somewhat difficult to spot the relation between the boxes and their respective nodes. Their system was developed in Java.

The Prototype Verification System (PVS) [20] is another proof assistant that provides capabilities for creating proofs and theories similar to IMPS. Its user interface is also Emacs based, and if the client system has Tcl/Tk (`http://www.tcl.tk`) installed, it offers the capability to graphically display proof trees. The analogous object to the IMPS sequent node is a node that is represented by the symbol "$\vdash$" in the PVS graphical representation of the proof tree and is followed by a connection to the proof command. If the latter exceeds a certain length, it is automatically abbreviated. The

user has the capability to click on the $\vdash$ symbol, and then a window pops up with text information about that sequent node.

The Tecton Proof System [14] is an experimental tool whose purpose is to construct proofs for formulas in the first order logic, as well as proofs for program specifications expressed using the Hoare's axiomatic proof formalism [11]. One of the interesting and related features that the system offers is proof visualization in the form of a series of pages with aligned text boxes, each of which is hyperlinked to relevant information. The interesting thing is how they manage the layout of these trees: they take into account the fact that a sequent node will never have more than one child (only one inference node). Thus, they combine each inference node with its parent sequent node to form a single node to save space.

CHAPTER 9

FUTURE WORK

There are multiple directions, in which future work can go. Apart from implementing the suggested new design, many new features can be added. Take positioning the graph in 3D space for one. It would be nice if the user has the capability to move certain branches (proof directions) in the deep space behind the graph, and then be able to toggle through them. This idea differs from the one used in the ACL2 proof visualization system (see Chapter 8) because rather than just representing the deduction graph as a 3D object that can be looked at from different angles, Panoptes shall allow the user to stack the proof branches on top of each other and then bring only the ones of interest to the front.

Furthermore, the multiscreen display was mentioned in section 5.3.4 on page 23, but regretfully that new technology was not available at the time of writing this thesis; therefore, testing on the multiscreen system has yet to be done. Since performance and speed are very important for this project, adjustments might need to be done, although preliminary testing on a system with 2 large screens, which act as one, was successfully completed and almost no lag was noticed.

Another very useful feature of the software would be the capability of comparing sequent nodes. That is, the program should be able to offer some numerical value, which represents the degree (i.e. percentage) of similarity. The ACL2 visualization system [1], mentioned in Chapter 8, has the capability to match the parse tree of

expressions to a certain degree. Panoptes should be able to offer similar functionality but be more open to different expression syntaxes. Speaking of which, future work should definitely include tuning the communication and parsing modules of Panoptes to work with other theorem provers.

Finally, the most ambitious plan includes expansion of Panoptes to completely replace the current Emacs-based user interface of IMPS. This is going to be exceptionally difficult, but in no way impossible. Difficult, because the user interface gives the user complete control over the development of proofs and theories and has many things to account for. For example, it maintains a constant connection with the concurrently running IMPS process and "consults" with it on the availability of commands the user can make. Also, it is responsible for updating IMPS with every choice made by the user—from changing the working theory to loading theorems and keeping track of proof steps. On the surface, it may not look all that complicated, but in reality the details that need to be taken care of are numerous.

CHAPTER 10

CONCLUSION

Proof assistants are software systems that allow the user to develop formal proofs by exercising high-level reasoning without having to worry about the large amount of low-level details. The user interfaces of these systems usually rely on the use of text to present the proof structures the user has developed. Often this text becomes so lengthy that it is confusing and difficult to comprehend and work with. That is why more effective tools for exploring such large proof structures are needed.

This work provides the requirements and design of a system called Panoptes that demonstrates what these tools need to provide. Inspired by the famous English idiom "*a picture is worth a thousand words*," Panoptes allows the user to explore deduction graphs created by the IMPS Interactive Mathematical Proof System in the form of graphical visualizations. A fully functional prototype of Panoptes has been implemented in OCaml. The prototype utilizes the powerful features offered by today's computer graphics technology. The result is a system that provides an easy and effective way of exploring these proof structures, complemented by a pleasant user experience.

CHAPTER 11

ACKNOWLEDGMENTS

First and foremost, I would like to thank my academic supervisor and mentor Dr. William M. Farmer. Not only did he introduce the world of computerized theorem proving to me, but he was a valuable and irreplaceable source of guidance for me throughout the complete process of developing this thesis and implementation, as well as during my whole stay at McMaster University. He made absolutely brilliant suggestions and helped me formulate and accomplish my ideas. Thank you, Bill!

I sincerely appreciate the time spent by the other members of my Examination Committee, Dr. Tom Maibaum and Dr. Spencer Smith, on reading this thesis and suggesting improvements that made the thesis much better. Also, special thanks to my colleague Pouya Larjani for his valuable programming tips, to Dr. Jacques Carette who read my thesis proposal and suggested OCaml for implementing the prototype, and to Dr. Wolfram Kahl for assisting me with the testing on the multiscreen machine.

I would like to thank my wife, Silvia, for her help, love and trust. Also, my daughter, Ema, has been a great stress reliever: her smiles, hugs, and love are almost magical!

I dedicate this work to my father, Grigor, and to my mother, Donka. I do not know another father who has fought so selflessly and hard to provide a better future for his family, and without my mother's advice and encouragement, I couldn't have made it this far. Mom and dad, I love you!

APPENDIX A—MODULE GUIDE

This appendix is separated into six sections. The first two sections (A.1 and A.2) list the anticipated and unlikely changes on the requirements of the system that might occur. The rest of the appendix contains a section on the module decomposition of Panoptes, a detailed description of each module, a uses hierarchy diagram, and a traceability matrix for easy identification of the modules that are responsible for each particular task.

A.1 Anticipated Changes

This section lists the changes on the software that are likely to happen. They are given names and numbers to facilitate matching them with the modules of the system later in the Traceability Matrix (Section A.7). A modification to one of these anticipated changes should ideally result in the need to change only one module, although the design is not optimized for that due to the rapid method of programming that was used for developing the prototype (also, recall that design and implementation occurred concurrently to the process of gathering the requirements).

AC_1 Keyboard Input:
Data structure and algorithms that provide the interface between the keyboard strokes and the system.

AC_2 Mouse Input:
Data structure and algorithms that provide the interface between mouse clicks and motion, and the system.

AC_3 Screen Display of Graphics:
Data structure and algorithms that provide means for displaying graphics on the screen.

AC_4 Screen Display of Text:
Data structure and algorithms that provide means for displaying text on the screen.

AC_5 User Interface:
Data structure and algorithms that create, support, and control the user interface.

AC_6 User Input Format:
The format of the input from the user, and the way to transmit it from user to system (e.g. by means of keyboard strokes, mouse clicks and movements, screen selection, etc.).

AC_7 Output Format:
The format of the output, and the way to deliver it to the user (e.g. graphics on the screen, file dump, shell, etc.).

AC_8 Graph Layout Engine:
The algorithm for generating a layout of the deduction graph on the screen.

AC_9 Format of the Input from the Proof Assistant:
The format of the data describing the deduction graph that the output module of the proof assistant creates.[1]

AC_{10} Dimensions:
Data structure and algorithms that manage the two dimensional appearance of the graph and its components on the screen.[2]

AC_{11} Deduction Graph Shapes:
Data structure and algorithms for drawing the shapes of the different elements of the deduction graph (e.g. arrows, different nodes, information boxes, etc.).

A.2 Unlikely Changes

This section lists the changes to the system that are unlikely to happen. It is important to understand that if the need to make any of these changes occurs in the future, it will inevitably result in the need to modify a large part of the system (possibly a large number of modules). All unlikely changes are labeled with "UC" and a number in subscript.

UC_1: The deduction graph is a bipartite graph.

UC_2: The edges in the deduction graph are directed.

UC_3: Cycles are allowed in the deduction graph.

UC_4: One of the two kinds of nodes will never have more than one parent (in the current case study, these are the inference nodes).

[1] In IMPS, the text version of the deduction graph is created by the Emacs-based user interface.
[2] Future expansion can include a 3D appearance.

A.3 Module Decomposition

Table A.1 shows the hierarchical breakdown of the modules. The top level decomposition consists of three modules:

- **Hardware-Hiding Module:** This module contains all parts of the system that need to be modified if some part of the hardware changes.
- **Behavior-Hiding Module:** This module contains all parts of the system that need to be modified if any of the required behavior (described in Section 5.2) changes.
- **Software Decision Hiding:** This module contains all parts of the system that implement algorithms and contain a chosen way of accomplishing tasks.

These top level modules are further broken down into smaller and more manageable units. Consequently, the "leaf" modules (the shaded cells) are the modules that were implemented and comprise the prototype of Panoptes.

Level 1	Level 2	Level 3
Hardware-Hiding Module	Input Module	Keyboard Input Module Mouse Control Module File Read/Write Module
	OpenGL Module	
Behavior-Hiding Module	Data Module	Parser Module Sqn_grabber Module TextGL Module Node Module Arrow Module Canvas Module Panoptes Module
	Drawing Module	
	User Interface Module	
Software Decision Hiding	Graph Layout Module	Automatic Layout Module Parser_dot Module
	Types Module	

Table A.1: Module Decomposition

A.4 Description of Modules

This section provides a detailed description of each module. Each such description is broken down into three parts: a label, a secret, and a service.

- **Label:** this part provides the label that is used to provide reference to the module in the Traceability Matrix (Section A.7).

- **Secret:** this part describes what the module hides from the rest of the system.
- **Service:** this part describes the functionality that is provided by the module.

Furthermore, the word "external" in the captions of some of the subsections that follow signify that the respective module is either imported from external libraries, or provided by the system, which means that there is no need for it to be implemented.

A.4.1 Keyboard Input Module (external)

- **Label:** M_1
- **Secret:** The data structure and algorithms for implementing the interface between the keyboard and Panoptes.
- **Service:** Collects the input from the user and communicates it to the other parts of the system.

A.4.2 Mouse Control Module (external)

- **Label:** M_2
- **Secret:** The data structure and algorithms for implementing the interface between the mouse and Panoptes.
- **Service:** Monitors the mouse for events, such as clicks and motion, and notifies the other parts of the system.

A.4.3 File Read/Write Module (external)

- **Label:** M_3
- **Secret:** The data structure and algorithms for accessing the file system.
- **Service:** Supplies the other parts of the system with an interface to files on the disk.

A.4.4 Window Initialization Module (external)

- **Label:** M_4
- **Secret:** The data structure and algorithms for initializing a window in the graphical environment of the operating system for the exclusive use of Panoptes.
- **Service:** Upon request, reserves the necessary resources, and by following internal procedures initializes and opens a graphical window that is at the disposal of the other parts of the system.

A.4.5 OpenGL Module (external)

- **Label:** M_5
- **Secret:** The data structure and algorithms for interfacing with the graphical hardware.
- **Service:** Provides the other parts of the system with a set of commands to access and control the graphical hardware of the computer system, which runs Panoptes.

A.4.6 Automatic Layout Module (external)

- **Label:** M_6
- **Secret:** Algorithms for generating a space-constrained two-dimensional layout of the deduction graph.
- **Service:** Provides a generated layout of the deduction graph in the form of positions on the screen for each node of the deduction graph.

A.4.7 Types Module

- **Label:** M_7
- **Secret:** Definitions of commonly used data types.
- **Service:** Provides a reference for the types to the type inferencing algorithms of the programming language.

A.4.8 Parser Module

- **Label:** M_8
- **Secret:** Data structure and algorithms for transforming the data input, supplied by the proof assistant (in this case this is IMPS), into internal Panoptes data structures. The module also hides these internal data structures, and also hides the operations for accessing and submitting queries to them.
- **Service:** Provides operations on the structure of the deduction graph, such as removal of nodes, addition of new nodes, traversing, obtaining information about nodes, statistics on the deduction graph.

A.4.9 Sqn_grabber Module

- **Label:** M_9
- **Secret:** Data structure and algorithms for detecting changes on selected files on the disk. Also, hides the algorithms for retrieving and storing of the detected new information in these files.
- **Service:** Provides the functionality to poll a specific file for changes to the rest of the system and retrieve any detected new information in it.

A.4.10 Parser_dot Module

- **Label:** M_{10}
- **Secret:** Data structure and algorithms for calling the layout module (see Subsection A.4.6) and parsing its output. Also, it abstracts the internal data structure used for keeping this information, as well as the algorithms for submitting and executing queries on the data structure.
- **Service:** Provides means for the system to run a deduction graph through the layout module in order to get a layout of the deduction graph. Provides methods for accessing the result during the time of drawing performed by the other parts of the system. This module can be globally viewed as the module that provides the layout of the deduction graph.

A.4.11 TextGL Module

- **Label:** M_{11}
- **Secret:** Data structures and algorithms for displaying text in an OpenGL graphical environment. Also, hides the chosen optimization algorithms used for increasing the runtime performance of the system.
- **Service:** Provides the other parts of the system with methods for displaying text in the form of graphical textures inside the OpenGL drawing canvas. Also, provides preliminary analysis and results about the eventual dimensions of a text block if it were to be converted in an OpenGL graphical texture format.

A.4.12 Node Module

- **Label:** M_{12}
- **Secret:** Data structure and algorithms for drawing a node in an OpenGL drawing canvas.

- **Service:** Provides a class, which can be instantiated and represents a node of the graph. The instantiated object has different methods that control the location and appearance of the node.

A.4.13 Arrow Module

- **Label:** M_{13}
- **Secret:** Data structure and algorithms for drawing a directed edge between two nodes.
- **Service:** Provides the system with a class, which can be instantiated to represent an arrow between two nodes. It contains methods for drawing the arrow on the screen, as well as keeping it connected at all times to its end points (the two connected by the arrow nodes).

A.4.14 Canvas Module

- **Label:** M_{14}
- **Secret:** Data structure and algorithms for most of the user induced operations in Panoptes. This includes zooming, collapsing, dragging and movement, etc.
- **Service:** Provides the other parts of the system with methods that can be called to achieve different transforming actions on the deduction graph. It directly controls many of the other modules (see Section A.5).

A.4.15 Panoptes Module

- **Label:** M_{15}
- **Secret:** Algorithms and machinery for initialization of a dedicated graphics window, as well as binding all keyboard shortcuts and mouse movements and clicks with the appropriate functionality of the system.
- **Service:** Serves as the starting point of system. Initializes the environment and provides it to the other parts of the system. Directs input events to the appropriate methods in the appropriate modules.

A.5 Uses Hierarchy

In [21] Dr. Parnas suggested that the statement "a program A uses a program B" means that correct execution of B is necessary for A to complete its task according to its specification. Therefore, there are situations, in which the correct functioning

of A is dependent on the correct implementation of B.

Following this principle and the decomposition of Panoptes into a few separate modules (see Section A.3), this section describes the dependence between these modules on each other, so that they can function properly. Also, it was already shown that each module is responsible for a different task. The complete "Uses Hierarchy" of Panoptes is shown in Figure A.1, where all such dependancies between the modules of the system are described in the form of a diagram.

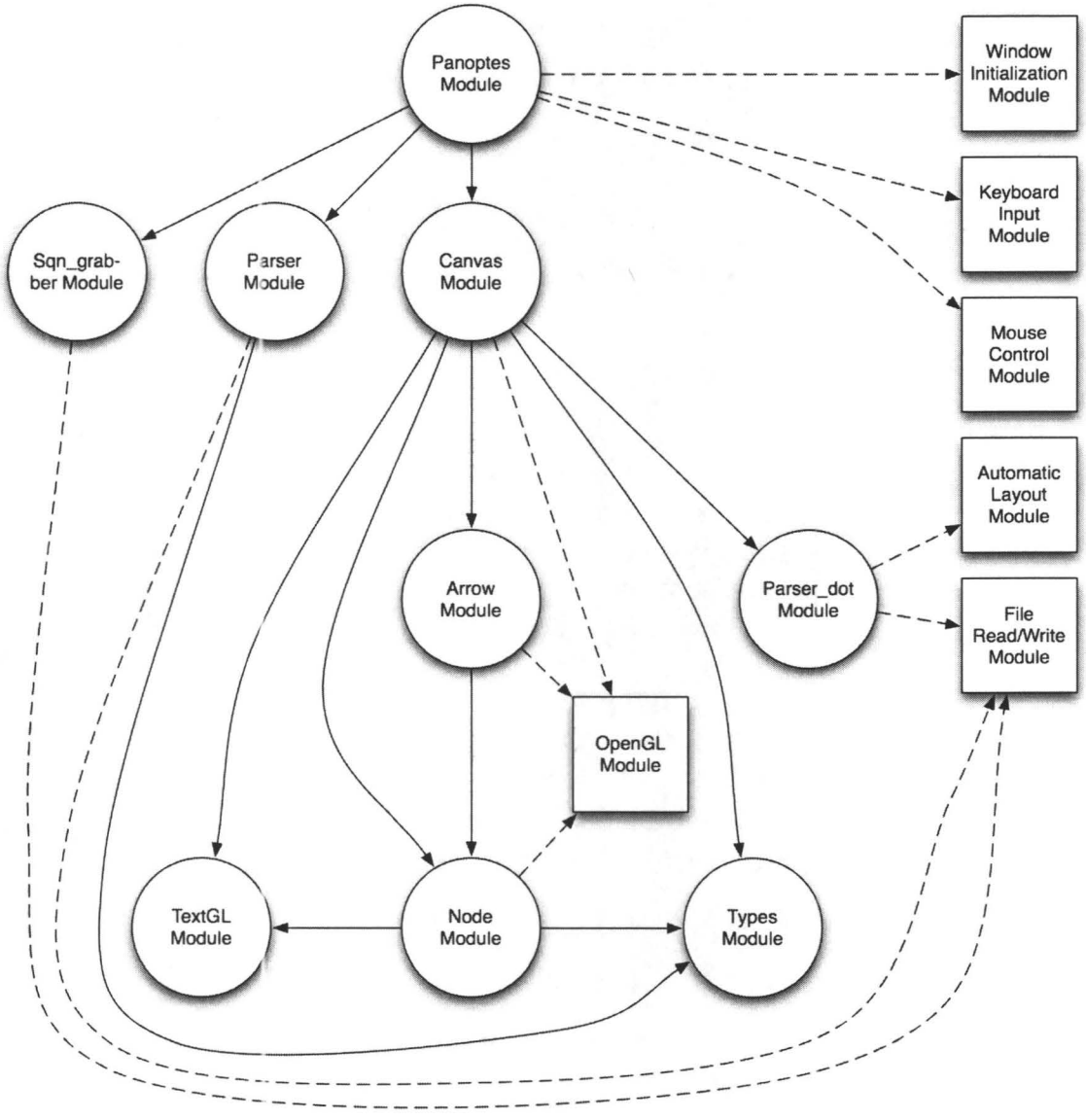

Figure A.1: Uses Hierarchy

The diagram contains a few different kinds of elements:

- **Circles** represent modules that need to be implemented.
- **Squares** represent modules that are provided by the system or are imported from external libraries.
- **Solid arrows** represent relationships (see below) between modules that need to be implemented.
- **Dashed arrows** are used for relationships between modules when one of them must be implemented and the other one is provided by the system or imported from an external library.

The arrows in the diagram represent dependency, i.e. the correct functioning of the module at the origin of the arrow depends on the correct functioning of the module pointed to by the arrow. Thus, an arrow that originates at **Module A** and points at **Module B** represents a relationship between the two modules, which means that **Module A** requires that **Module B** is implemented and functions properly for the former to work. Furthermore, the **Module A** can call the access methods of **Module B** (see Section A.4 for the access methods of all modules).

A.6 Summary of Requirements

For convenience, two lists are provided below: one containing all functional requirements, and one containing all nonfunctional requirements, which were stated and defined in Chapter 5. The items in the functional requirements list start with an "R" and the number of the functional requirement in subscript, and the items in the nonfunctional requirements list start with an "NF" and the number of the nonfunctional requirement in subscript. These labels are used in the Traceability Matrix that follows in the next section. Also, each requirement label in the list is complemented by a short description of what it represents. For more detailed information regarding a particular requirement, the reader should refer to the respective page number of that requirement that is also provided below.

A.6.1 List of Functional Requirements

R_1 (page 17): Nodes must be visualized on the screen.

R_2 (page 17): Nodes must be visually different from each other.

R_3 (page 17): Special nodes must be colored differently.

R_4 (page 18): Sequent nodes must be labeled with their IMPS names.

R_5 (page 18): Inference nodes must be named after the respective command they represent in IMPS, and also must be numbered to provide information about repetitions of the same command.

R_6 (page 18): The user must be able to rename the labels of the nodes.

R_7 (page 19): Each box must have an information box with detailed information, which can be opened or closed.

R_8 (page 19): The program should suggest a good initial layout of the deduction graph.

R_9 (page 20): The user must be able to manually rearrange the layout of the deduction graph.

R_{10} (page 20): The user should be able to do targeted zooming on any part of the graph.

R_{11} (page 20): The user should be able to collapse parts of the graph into newly created nodes without destroying the properties of the graph.

R_{12} (page 21): The user should have a means of examining the collapsed boxes to learn what they represent.

R_{13} (page 21): The collapsed boxes should be labeled in a way that provides the user with a hint regarding the information hidden by them.

R_{14} (page 21): All user induced operations should be reversible.

R_{15} (page 21): A help screen with available commands should be available to the user.

A.6.2 List of Non-Functional Requirements

NF_1 (page 22): The tool should be easy to learn.

NF_2 (page 22): The tool should be easy to use.

NF_3 (page 22): The user should be automatically protected from making errors and illegal operations on the deduction graph.

NF_4 (page 23): Target hardware: a modern graphics card.

NF_5 (page 24): A good choice of optimized algorithms for computations should be made.

NF_6 (page 25): Centralization of the part of the system that is responsible for accepting and processing the input data.

NF_7 (page 25): The target operating systems should be supported.

NF_8 (page 25): The design should account for the ability of Panoptes to run on large screen walls.

NF_9 (page 25): The installation should be easy.

A.7 Traceability Matrix

	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}	D
AC_1	✓															
AC_2		✓														
AC_3				✓	✓											
AC_4											✓					
AC_5														✓	✓	
AC_6															✓	
AC_7												✓	✓			
AC_8						✓				✓						
AC_9			✓					✓	✓							
AC_{10}														✓		
AC_{11}												✓	✓			
R_1				✓	✓							✓	✓			
R_2					✓							✓				
R_3					✓			✓				✓				
R_4								✓			✓					
R_5								✓			✓					
R_6	✓	✓						✓						✓	✓	
R_7	✓	✓						✓	✓		✓	✓		✓	✓	
R_8			✓			✓				✓			✓			
R_9	✓	✓												✓	✓	
R_{10}	✓	✓												✓	✓	
R_{11}	✓	✓						✓						✓	✓	
R_{12}	✓	✓									✓			✓	✓	
R_{13}														✓		
R_{14}								✓						✓		
R_{15}														✓	✓	
NF_1															✓	
NF_2															✓	
NF_3														✓	✓	
NF_4					✓											
NF_5								✓						✓		
NF_6								✓								
NF_7	✓	✓	✓													
NF_8				✓	✓									✓		
NF_9							✓									✓

Table A.2: Traceability Matrix

Table A.2[3] provides the Traceability Matrix, which shows the coupling of the anticipated changes, the functional requirements, and nonfunctional requirements with the modules of the system. The labeling of the modules comes from Section A.4, and the labels of the anticipated changes come from Section A.1. The labels for the functional and the non-functional requirements come respectively from Sections A.6.1 and A.6.2.

[3]The last column named "D" represents the instructions for installing the Panoptes prototype, which are provided in Section 7.3 on page 52.

APPENDIX B—DOCUMENTATION OF THE IMPLEMENTATION

This appendix contains a detailed method description for each module of the system (except the modules that are either imported from external libraries or are provided by the system[4]). A separate section is dedicated for each module, which contains information on all implemented methods and their types within the module. In addition, all used variables are included with descriptions of their purposes. For best understanding and clarity, the reader is advised to have the source code available as the names of the arguments passed to the methods are self-explanatory in most cases.

[4]These external modules are indicated by the word "external" that follows the module's name in Section A.4 on page 72.

B.1 Module Sqn_grabber

```
class sqn_grabber : string ->
  object
```

`val mutable last_modtime : float`

This variable holds an indicator of the time format, which is platform dependent (in OS X and Linux it is in the form of a real number). It is used for comparison with the current timestamp of the file that is monitored for changes. Every time the file changes, the object of this class will detect this as an event and will update this variable accordingly.

`method stop : unit -> unit`

This method stops the thread and removes the backup file from the disk (the file, which is used for storing any detected new data).

`method poll : unit -> bool`

This method "polls" the file that is being monitored. If an update is detected, then it creates a backup of the newly introduced data. Also, it transmits a notification message to the `Panoptes` module, letting it know that it should notify the user about the detection of new data. Consequently Panoptes displays an indication on the screen that prompts the user to press the "UPDATE" shortcut in the user interface.

```
  end
```

B.2 Module Types

```
type kindT =
  | SQN
  | INF
```

This type is used to indicate whether something represents a sequent node or an inference node.

```
type specialT =
  | Nil
  | Repeated
  | Grounded
  | Box
```

This type is provided for use by the parser. The different kinds of nodes in the deduction graph are labeled differently. Recall that IMPS automatically labels repeating and grounded nodes in its text representation of the deduction graph. The "Box" value is used later in the implementation to signify a node that represents a collapsed part of the graph.

```
type nodeT =
  | Node of kindT * string * specialT * nodeT list
```

This is a recursive type that is used for representing nodes in the graph. Each record of this type consists of a tuple of elements: the kind of the node, its name (or label), its special characteristics, and a list of other nodes it is connected to. This resembles the design of a linked list.

```
type sqnT = (int * specialT * string) list
```

This is used to represent a collection (or list) of nodes. This type is not very important, but its presence makes implementation more convenient and less confusing. When programming in OCaml it is important to specify the types in advance, so that the built in type inference system of the compiler produces performs better.

```
val pi : float
```

The number Pi to be used throughout the modules of Panoptes that import this module. Here it is calculated to a great enough precision to accommodate some graphical calculations used in the other modules.

B.3 Module TextGL

```
class font :
  object
```

```
    val mutable chars :
      (int, ([ 'rgb ], [ 'ubyte ]) GlPix.t * int * int) Hashtbl.t
```

A hash table that is indexed by an integer that represents a character of the alphabet. Each record contains a tuple, consisting of the graphical image of that character and its width and height in pixel units.

```
val mutable chars_texture :
 (int, GlTex.texture_id * int * int) Hashtbl.t
```

A hash table that is indexed by an integer that represents a character of the alphabet. Contrary to the hash table described above, this hash table does not keep an image but the pre-generated OpenGL texture for the character. This is needed to skip the time consumed for generating a new OpenGL texture each time such texture is needed. Such need usually occurs during synchronization of the deduction graph in Panoptes with the one in IMPS. As stated many times already, speed performance in Panoptes is critical for a pleasant user experience.

```
method initialize : unit -> unit
```

This method initializes the font by creating two empty hash tables (as described above) that will contain the images and textures of all characters in the alphabet. Then the method calls the `make_image` method (see below) that is used to populate these hash tables.

```
method private input_binary :
  int -> ([ `rgb ], [ `ubyte ]) GlPix.t * int * int
```

Opens a pregenerated binary image file that contains contains a sequence of all characters in the alphabet. Then it loads it into an internal data structure that is returned along with its dimension in pixel units.

```
val mutable char_width : int
val mutable char_height : int
```

These are two global variables that respectively hold the width and the height of the characters in the alphabet (measured in pixel units).

```
method private make_image : unit -> unit
```

Loads the font by traversing the data structure that is returned by `input_binary` method, and extracts each individual character from it. The data describing each character is then stored in the hash table `chars` for further use.

```
method get_char : int ->
 ([ `rgb ], [ `ubyte ]) GlPix.t * int * int
```

This is a public method that returns the pregenerated OpenGL texture of the requested by the argument character.

```
method get_text : string ->
 ([ `rgb ], [ `ubyte ]) GlPix.t * int * int
```

Similarly to the method described above, this is a public method that generates and returns an OpenGL texture that represents the supplied by the argument string of characters.

```
method private clean_text_newlines : string -> string * int
```

This method is used to clean the supplied by the argument string from the characters used to represent new lines. These characters are replaced by the symbol '@', which facilitates the formating of the text (shift new lines down and to the left when generating a new texture from a string of characters) done by other client methods.

```
method private add_texture :
  int -> ([ `rgb ], [ `ubyte ]) GlPix.t * int * int -> unit
```

This private method is used to load the texture that is associated by the argument into the video memory of the graphical card. This is extremely important as the hardware can then use it immediately when needed instead of sending queries to the main memory. This pre-loading of textures method proved to be very efficient through direct testing done with Panoptes.

```
method printGL : string -> unit
```

This method displays a line of text directly in the OpenGL drawing window (see `printGL_block` below for more detailed description of this process).

```
method private find_longest_line : string -> int
```

This private method is used to preliminary analyze a block of text in order to determine the length of the longest line contained in it. This is used to correctly determine the right amount of memory that needs to be reserved for the image that will eventually represent this block of text.

```
method calc_ratio_block : string -> float
```

This method calculates the ratio between the width and the height of the block of text supplied by the argument. For example, if the text consists of 4 lines, the longest of which contains 10 symbols, then the ratio will be $\frac{10}{4} = 2.5$. This is used by the objects of `Node` class in order to do the right calculation for proportionate scaling and positioning of the node and its components (the frame and the label).

```
method printGL_block : string -> unit
```

This is a very popular public method as it is used extensively by other parts of the system. It allows immediate display of the text provided by the argument in the OpenGL window. Therefore, this method is not only responsible for calling the other private methods associated with this process (such as the method for cleaning the text from newlines and the method for generating a texture in case when it does not already exist), but also it is responsible for calling the OpenGL procedures to draw the text on the screen. The reader should know that prior to calling this method, the position of the text to be displayed should be set in the OpenGL engine.

```
method get_texture : int -> GlTex.texture_id * int * int
```

This method takes an ASCII code of a character (supplied by the argument), and then returns the index number that is associated with that character in the hash table that holds the pregenerated OpenGL textures.

```
end
```

B.4 Module Parser

```
class graph_data_class : string * string * string ->
  object

    val mutable sqn_data :
      (String.t, Types.specialT * string * 'a list) Hashtbl.t
```

This hash table holds the details for the information boxes of each sequent node. The table is indexed by the text name of the sequent node in IMPS, and it contains a tuple of the following elements: one indicating whether the node is special (repeated, grounded, etc.) and one that holds a string of text that contains the details of the node. The third element of the tuple is not used. It is added for convenience and eventual future expansion of functionality.

```
    val mutable sqn_list :
      (String.t, Types.specialT * String.t list * String.t list)
      Hashtbl.t
```

The elements of this hash table comprise the set of all sequent nodes in the deduction graph. They are indexed by their IMPS names, and the tuple of information contains their specialty and two separate lists of strings: one for all children of the node and one for all parents of the node.

```
val mutable inf_list :
  (String.t, Types.specialT * String.t list * String.t list)
  Hashtbl.t
```

Similarly to `sqn_list` this hash table stores all inference nodes in the deduction graph. They are are indexed by their names in IMPS and their assigned numbers in the parser method of this class.

```
method remove_node : String.t -> unit
```

This method removes the existence of the node indicated by the parameter `name`. After this method completes its operation, the indicated node does not appear in any of the data structures of this class.

```
method graph_stats : int * int * int * int
```

This method returns four integers: the number of sequent nodes, the number of inference nodes, the number of grounded nodes, and the number of nodes that are repeated and form a cycle in the deduction graph.

```
method private make_dot_string : string
```

This method returns a string that represents the deduction graph in a format that is recognizable by the Graphviz external software. Recall that this software is used for generating the initial layout of the deduction graph in Panoptes.

```
method update : unit
```

This method groups the calls to all other methods of this class to achieve a "reset and recollect" operation regarding all information that describes the deduction graph.

```
method get_node_parents : String.t -> String.t list
```

This method returns a list of strings. These strings contain the names of all parents of the node indicated by the parameter of the call.

```
method find_the_way_back : String.t -> String.t list
```

This method returns a list of node names in the form of strings. These nodes comprise the path from the indicated by the parameter node, all the way to the root node in the deduction graph (the oldest sequent node in the graph, a.k.a. the main goal).

```
method get_node_children : String.t -> String.t list
```

This method, similarly to `get_node_parents`, returns a list of all children of the indicated by the parameter node.

```
method find_the_way_forward : String.t -> String.t list
```

This method returns a list of nodes. These nodes represent all successors of the indicated by the parameter node.

```
method find_path_to_from : String.t -> String.t -> String.t list
```

This method returns a list of nodes, which comprise the path between the two nodes indicated by the two arguments. If there is not a direct path between these two nodes, then the resulting list will remain empty.

```
method get_node_details : String.t ->
 Types.specialT * Types.kindT * string
```

This method accepts the name of the node as an argument, and returns a tuple consisting of three things: the specialty of the node (repeated, grounded, etc.), the kind of the node (sequent or inference node), and its detailed information (supplied in the form of a string).

```
method write_dot_file : string -> string -> unit
```

This method writes to the disk the result from the method mentioned above: `make_dot_string`. The resulting file on the disk will be used in a subsequent step as an input to the external program GraphViz that is used to generate the initial layout of the deduction graph in Panoptes.

```
method private sqn_clean : string -> String.t
```

This private method replaces the letter sequence "IMPS_SQN_" with "SQN_" in the supplied by the argument string.

```
method private inf_clean : string -> String.t
```

This private method adds a number to the name of an inference node. This number shows how many times the inference rule that is represented by that node has been used in the deduction graph until that point.

```
method print_dg : unit
```

This method is used only for debugging purposes. It implements a textual pretty printer of the deduction graph, the result of which is very similar to the format used internally in the IMPS user interface. It displays its output in the active shell window that was used to start Panoptes.

```
method add_new :
  Types.kindT -> String.t -> Types.specialT ->
  String.t list -> unit
```

This method adds a new node to the deduction graph. It can be considered polymorphic in the sense that it can be used for adding both kinds of nodes—sequent and inference nodes. Also, if known, a list of children of the node to be added can be supplied. If such list is not available at the time of calling this method, then the `list` argument can accept an empty list.

```
method add_new_to_parent_list :
 Types.kindT -> String.t -> String.t -> unit
```

This method adds a parent node to the list of parents of the indicated by the argument node. Again, the kind of the target node (the one, whose list of parents is to be updated), must be passed as an argument.

```
method add_new_to_list :
 Types.kindT -> String.t -> String.t -> unit
```

Similarly to the method mentioned above (`add_new_to_parent_list`), this method adds a new child node to the list of children of the indicated by the argument node.

```
method private read_dg_file : string -> unit
```

This is a very lengthy method, whose purpose is to read the IMPS file that describes the structure of the deduction graph. After that is done, it parses the information and with the help of the other methods builds and populates the internal data structure used for storing the deduction graph. For the parsing part, a lexer is created at the beginning, which separates the input into individual tokens. The parser then matches these tokens according to the rules of the deduction graph abstract syntax tree, and populates the structure recursively.

```
method private duplicate_DG_file_no_hyphens : string -> unit
```

In order to protect the parser from confusion when a few known to be problematic special characters are present, all such characters are replaced by equivalent in their meaning other strings. Then the file can be used safely by the lexer and parser in the `read_dg_file` method of this class.

`method print_sqn : unit`

This is a pretty printer for displaying the details of all sequent node in the command shell. This method is used mainly for debugging purposes.

`method private duplicate_SQN_file_no_formfeed : string -> unit`

The file that contains the details of the sequent nodes is returned by IMPS in a special format. Basically, IMPS uses the formfeed character to separate the information of the sequent nodes from each other, which confuses the parser. Consequently, this method was created and used to replace all occurrences of this character in the file with another "dummy" character that is used to indicate the separation points.

`method private read_sqn_file : string -> unit`

This method creates a string from the file that holds the details of the sequent nodes. Consequently, the method makes calls to the parser in an incremental manner by simultaneously populating the data structure (represented by the hash table `sqn_data`).

`end`

The arguments for instantiation of this class consist of three strings. The first argument provides the system path to the current directory, which is needed for dealing with a compatibility issue on some Mac systems. The second and third string provide the filenames for the IMPS deduction graph and the `sqn_grabber` storage files respectively.

B.5 Module `Node`

```
class node : string * (Types.specialT * Types.kindT) *
 GlTex.texture_id * string *

  (float * float) * float * float * float * float * float *

  < calc_ratio_block : string -> float; printGL_block :
 string -> 'a; .. >

  Pervasives.ref * int * float ->
  object
```

```
method get_all_specs :
  (float * float) * float * float * float *
  float * float * float * float *
  float * float * float * float
```

This method returns a tuple of values. Each value represents a certain piece of information about the node. The argument names, which can be found in the source code, are self-explanatory.

```
method get_kind : Types.kindT
```

Returns the kind of the node represented by the particular object of this class. The kind can be a sequent node or an inference node.

```
val mutable name_texture : GlTex.texture_id
```

A variable that represents a pointer to the OpenGL texture containing the display name of the node.

```
val mutable x : float
val mutable y : float
val mutable z : float
```

The x, y, and z coordinates of the initial layout location of the node.

```
val mutable width : float
val mutable height : float
```

The width and the height of the node.

```
val mutable content_text : string
```

A string that contains the detailed information of the node. It represents the contents of the associated with this node information box.

```
val mutable content_ratio : float
```

The ratio between the width and the height of the information box if it were to be drawn to the OpenGL window.

```
method rename_node : GlTex.texture_id -> string -> int -> unit
```

This method renames the node and then calls all other appropriate methods that update the variables, which describe the content box (e.g.˜the content box ratio, text, etc.

```
val mutable rx : float
val mutable ry : float
val mutable rz : float
```

The x, y, and z distances from the initial layout location of the node. These can also be understood as the relative positions to the permanent position of the node. This is used for implementing the ability to revert back to the initial layout of the deduction graph upon request by the user.

```
method private get_rx : float
method private get_ry : float
method private get_rz : float
```

This method makes the previous three positions available to other parts of the system.

```
val mutable hovered_flag : int
```

A flag that indicates whether the mouse cursor is hovering over the node at any particular moment.

```
val mutable selected_flag : int
```

A flag that indicates whether the content box is currently open (if it is visible).

```
val mutable path_flagged : bool
```

A flag that indicates whether the node participates in a path visualization procedure.

```
val mutable cont_w : float
```

The initial and default width of the information box associated with the node.

```
val mutable center_box_x : float
val mutable center_box_y : float
val mutable center_box_z : float
```

Coordinates of the current position of the content box.

```
method get_x : float
method get_y : float
method get_z : float
```

These three methods return the current global position of the node in the OpenGL window. They take into account the initial layout position, as well as the relative position of the node.

```
method get_width : float
method get_height : float
```

These methods return the width and the height of the node in OpenGL distance units.

```
method calc_x : float -> float
method calc_y : float -> float
```

These two methods convert the supplied by the arguments coordinates into global positions in respect to the current layout and node transformations (location translation, relative positions, scaling, etc.).

```
method get_name : string
```

This method returns the string name of the node. It can be further used for identification of the node in the other parts of the system and their data structures.

```
method get_box_x : float
method get_box_y : float
method get_box_z : float
```

These methods make the coordinates of the content box position available to the other parts of the system.

```
method get_path_flag : bool
method set_path_flag : bool -> unit
```

These two methods respectively return and set the flag that indicates whether the node participates in a path visualization procedure.

```
method draw_content_box : unit -> unit
```

This method draws (displays) the information content box of the node on the screen (if it is in a visible state).

```
method is_selected : bool
```

This method can be used by other parts of the system to learn whether the content box of the node is currently open (visible).

```
method draw : unit -> unit
```

This is the method that draws the node in the OpenGL drawing space.

```
method reposition_box : float * float * float -> unit
```

This method allows moving of the content box to a new position in the OpenGL drawing space.

`method reposition_node : float * float * float -> unit`

This method allows moving of the node to a new position in the OpenGL drawing space.

`method rdraw : float * float * float -> unit`

Same as above, but instead of a new position, the arguments of this method represent values of the distances from the current position of the node to the original location of the node in the initial layout generated by GraphViz.

`method hovered : int -> unit`

Sets the flag when the mouse cursor is over the node and unsets it when mouse leaves.

`method toggle_selected : unit -> unit`

Toggle the flag that indicates whether the content box is open or closed.

`method scale_content_box : float -> unit`

Scales the content box by a certain amount. This creates the effect of zooming for the content box. Notice that zooming of the content boxes is achieved via different method than the zooming performed on the whole graph: the graph is zoomed by moving it closer or further away from the viewer.

`method change_cont_ordering : float -> unit`

This method is used to pull the content box above all other content boxes of all other nodes. This functionality is useful in the events when content boxes overlap and the user wants to focus on this particular content box. Of course, the higher the order (supplied by the argument), the lower priority there is for this content box to be displayed, which is a machinery to facilitate the design of the `Canvas` class.

`end`

There is one instantiation of this class for each node in the deduction graph. Referring to the source code and the names of the arguments, it can be understood what each of them means as the names of the arguments are meaningful.

B.6 Module Arrow

```
class arrow : Node.node * Node.node * (float * float) list ->
  object
```

`val origin : Node.node`

A reference pointer to the object representing the originating node of the arrow.

`val destination : Node.node`

A reference pointer to the object representing the destination node of the arrow (the node that the arrow points at).

`val mutable pairs : (float * float) list`

This is a list of control points for the edge. This method consequently became obsolete: these points are used for drawing bezier curves, which were disabled at the point in the implementation when the functionality for allowing manual repositioning of the nodes in the graph was introduced. Before that, these control points were being generated upon deduction graph initialization by the program that generates the automatic layout for the graph (Graphviz).

`method get_origin : Node.node`

`method get_destination : Node.node`

These two methods return the reference pointers to the objects that represent the origin and destination nodes respectively.

`method draw : unit -> unit`

This method draws (displays) the arrow in the OpenGL drawing space.

```
  end
```

This class is instantiated to represent an edge in the deduction graph in Panoptes. Complying with the properties of the graph, each edge is associated with two different nodes.

B.7 Module Parser_dot

```
class parser_dot :
  object
```

```
    val mutable nodes : (string, float * float * float * float)
      Hashtbl.t
```

A variable that represents a hash table with all of the nodes (indexed by their string names). Each record contains four numbers: the x and y positions, the width and the height of the respective node.

```
    val mutable edges : (string * string, (float * float) list)
      Hashtbl.t
```

A variable that represents a hash table with all of the edges. Each edge is indexed by a tuple of two strings: the string names of the originating and destination nodes. Each record consists of a list of tuples of two real numbers. Each tuple represents a control point of the position of the edge. Thus, the first and last points will coincide with the positions of the two connected nodes, while the pairs in between represent control points that can be used for building Bezier curves.

```
    method get_nodes : (string, float * float * float * float)
      Hashtbl.t
    method get_edges : (string * string, (float * float) list)
      Hashtbl.t
```

These two methods make the hash tables that keep the information about all nodes and edges of the deduction graph available to the other parts of the system.

```
    method private read_dot_file : string -> unit
```

This method implements the parser for the ouput result, generated by the automatic layout generator Graphviz.

```
    method process_dot : string -> unit
```

This method invokes the automatic layout generator Graphviz.

```
  end
```

This class provides the parser for the output generated by the layout program Graphviz. It invokes this program and then processes its result. Contrary to the `Parser` module, which deals with parsing the IMPS ouput and builds the internal structure that represents the deduction graph, this module collects the information about the actual locations of all nodes and edges.

B.8 Module `Canvas`

```
class view : < add_new : Types.kindT ->
              String.t -> Types.specialT -> String.t list -> unit;
    add_new_to_list : Types.kindT -> String.t -> String.t -> 'a;
    add_new_to_parent_list : Types.kindT -> String.t ->
  String.t -> unit;
    find_path_to_from : String.t -> String.t -> String.t list;
    find_the_way_back : String.t -> String.t list;
    find_the_way_forward : String.t -> String.t list;
    get_node_children : String.t -> String.t list;
    get_node_details : String.t -> Types.specialT *
  Types.kindT * string;
    get_node_parents : String.t -> String.t list;
    remove_node : String.t -> unit; .. >
  Pervasives.ref ->
  object
     val mutable screenWidth : float
     val mutable screenHeight : float
```

These two variables comprise the current resolution of the OpenGL window. Upon initial initialization, the size is 800 pixels by 600 pixels by

default (although the user is allowed to resize the window without limitations).

```
val fovy : float
```

This variable holds the field of view (FOV) angle in degrees. Refer to the OpenGL documentation of the Implementation chapter of this thesis to find more explanation about FOV and how it works.

```
val mutable x1 : float
val mutable y1 : float
val mutable z1 : float
```

These variables specify the coordinates of the absolute center of the space (this space can be viewed as the complete universe, in which the deduction graph resides). The location of every single component from a mere point to a large and complicated shape has certain coordinates that are relative to the coordinates represented by these three variables. Of course, since we do not utilize the ability to move this center of space to achieve certain effects in Panoptes, the coordinates are conveniently reset to zeroes.

```
val init_z : float
```

The initial `z` coordinate of the deduction graph. It must be deep in space to be visible (away from the user) in order to be visible and within sight (the sight is defined by the FOV and other parameters, which are described later).

```
val mutable nodes : (String.t * Node.node) list
```

A list of tuples. Each tuple contains a node name (in string format) and a reference to a created object of the `Node` class, which is associated with that node. This list comprise all nodes of the deduction graph, which were created during initialization and are available for the program for direct OpenGL display.

```
val mutable arrows : Arrow.arrow list
```

A list of all created arrows (edges). Similarly to the previous list, this list contains all arrows in the deduction graph, which were created during initialization and are available for direct OpenGL display.

```
val mutable parser_dg :
  (< add_new : Types.kindT ->
               String.t -> Types.specialT -> String.t list ->
```

```
            unit;
    add_new_to_list : Types.kindT -> String.t -> String.t -> 'a;
    add_new_to_parent_list : Types.kindT -> String.t ->
    String.t -> unit;
    find_path_to_from : String.t -> String.t -> String.t list;
    find_the_way_back : String.t -> String.t list;
    find_the_way_forward : String.t -> String.t list;
    get_node_children : String.t -> String.t list;
    get_node_details : String.t -> Types.specialT *
    Types.kindT * string;
    get_node_parents : String.t -> String.t list;
    remove_node : String.t -> unit; .. >
  as 'b)
 Pervasives.ref
```

Instantiation of `Parser`. In fact notice that this object was passed to the current class upon initialization of this class. Therefore, this variable keeps a reference pointer to object that was already created.

```
val mutable dedotter : Parser_dot.parser_dot
```

A new instantiation of `Parser_dot` that will be used for running the deduction graph through the layout generation program GraphViz. Notice how the current class serves as a bridge between all other classes. In this case, the previous `parser_dg` will collect all information that describes the deduction graph. The current class will then retrieve this information and send it to `dedotter` for processing, which in turn will return the suggested by Graphviz locations of all nodes.

```
val mutable selection_names : (String.t, int) Hashtbl.t
```

This is needed by OpenGL in order to implement the selection technique. Usually, each object in the window is assigned a "selection value" upon instantiation. A special function then prompts OpenGL to run an imaginary linear ray from a certain point in space towards a certain direction. When this ray hits a target, the selection value of that target is returned. This value can then be matched against the tuples of this hash table to discover the name of the object that was hit. Of course, this object can be any possible element of the deduction graph that has a selection name assigned to it. In our case, such objects are the nodes and the information boxes, although the arrows are assigned a unique, but the same for all arrows, for facilitating the process of debugging. This selection values technique is hereby used to detect the object the user clicks on.

```
val mutable locked_node : bool * String.t * float *
  float * float
```

When the user presses down and holds the left mouse button over a certain node (notice that this class provides the functionality, while the actual event listeners are implemented in the `Panoptes` class), then the boolean of this variable becomes TRUE until the user releases the button. The other three values keep the 3D coordinates of the node at the moment the mouse button was pressed. Other methods, described below, access the values of this tuple every time a new screen frame must be drawn. If the first element is true, these other methods perform appropriate actions.

```
val mutable locked_dg_drag : bool * float * float
```

Similar to the above variable, this variable signifies a lock on the whole deduction graph. Its purpose is mainly used to notify the other methods that the user is dragging the graph with the mouse.

```
val mutable content_box_ordering : (String.t, float) Hashtbl.t
```

This variable keeps the `z` coordinates of all open (visible) content boxes. As such, some content boxes will be closer to the screen while others will be further away. This is important to achieve control over the focus of the content box that the user wants to be completely visible, especially when the content boxes are arranged in an overlapping manner.

```
val mutable added_arrows :
  (Arrow.arrow * String.t * String.t) list
```

This variable keeps a list of all newly added arrows. Such arrows are usually added when a collapsing procedure is performed. Recall the Design chapter, which stated that all collapsing procedures produce new nodes. It is logical that if there is a new node, it will be connected to other nodes through new links (in this thesis we usually call these links arrows). Furthermore, to facilitate the reversibility of all user induced actions (as requested by the Requirements chapter), these new arrows are conveniently stored in this separate list. Thus the original arrangement of the graph is always preserved.

```
val mutable collapsed :
  (String.t,
   String.t * (String.t * String.t list * String.t list) list *
   String.t)
  Hashtbl.t
```

A hash table that keeps information about all created by that point collapsed nodes. Each element of the hash table contains information about the nodes it hides, the arrows it is associated with, as well as other useful information.

`method reset_all_data : unit`

This method resets all variables of this class. It is called when the user decides to reset Panoptes, which usually happens when the development of a new proof is started in IMPS.

`method fit_to_screen : unit -> unit`

This method resets the location and zoom level of the deduction graph to the screen according to the generated initial layout. This is useful when the user manually relocates the deduction graph and later decides to revert back and fit everything in the screen.

`method private create_DG : unit`

This method builds the deduction graph. It creates instances of `Node` and `Arrow` for each graph node and edge. Verbose information is displayed in the command shell during this process, as this may sometimes take more than a second for graphs that contain hundreds of nodes. Additional things that are accomplished by this method include calculation and initialization of the initial ordering of content boxes, as well as assignment of selection values to all components of the deduction graph (the notion of selection values was described earlier in this section).

`method private is_it_in_collapsed : String.t -> bool`

The argument of this method is a string that holds the name of particular node. The return result of the method is a boolean value that is set to `true` when the supplied by the argument node is a part of a collapsed part of the graph.

`method private draw_DG : unit`

This method displays the deduction graph on the screen. When there is activity, such as dragging of objects on the screen, zooming or other visual movement, this method is called approximately 30 to 60 times per second (depending on how fast the graphical processing unit (GPU) is).

`val font : TextGL.font Pervasives.ref`

This variable holds a reference pointer to the instantiation of the TextGL class. Only the reference to this object is kept here, because it is later passed and used by the all instances of the `Node` class to draw the text labels of each node.

```
method add_node :
  String.t ->
  Types.specialT * Types.kindT ->
  float * float ->
  float -> float -> float -> float -> float -> string ->
  int -> float -> unit
```

This method instantiates a new object of the `Node` class. Then the method includes the newly created object in all relevant data structures and processes of the current class.

```
method add_arrow : Node.node -> Node.node ->
  (float * float) list -> unit
```

This method, similarly to the method described above, instantiates a new object of the `Arrow` class, and then includes it in all relevant data structures and processes of the current class.

```
method get_arrow : String.t -> String.t -> Arrow.arrow
```

This method returns the reference pointer to an `Arrow` object. This object represents the arrow that connects the supplied by the two arguments of the method nodes in the deduction graph.

```
method new_texture : String.t -> GlTex.texture_id * int * int
```

This method takes the string that is provided by the argument and creates a new OpenGL texture that can be used by OpenGL to display that string. Also, the method loads the texture into the video memory in order to make it available for immediate use upon need.

```
method get_node : String.t -> Node.node
```

This method retrieves the reference pointer to the object of class `Node` that is associated with the node represented by the string in the argument.

```
method get_hit_object_name : int -> String.t
```

This method matches the integer that is supplied by the argument with a selection value (described earlier). This is useful for finding the string name of the object that is assigned the selection value.

```
val mutable path_flag_list : String.t list
```

This list holds the string names of all nodes that participate in a visualization of a path between certain nodes in the deduction graph. It is used by other methods as a reference on which parts of the graph must be highlighted.

`method private clear_path_hit : unit`

This method resets (nullifies) the `path_flag_list` variable described above.

`method show_back_path_hit : int -> unit`

Based on the supplied by the argument selection value, which is possibly associated with some node, this method populates the `path_flag_list` that was specified above with all nodes that are traversed in order to reach the top node in the deduction graph (the main goal).

`method show_forward_path_hit : int -> unit`

Similarly to the `show_back_path_hit` method described above, this method populates the `path_flag_list` with all nodes that are traversed in order to reach all leaves in the graph by starting at the node associated with the supplied by the argument node selection value.

`val mutable collapse_selection : bool * String.t * String.t`

In order to collapse a section of the deduction graph, the user needs to select two nodes that represent the endpoints of the part to be collapsed. This variable holds the string names of these two nodes, which are stored in a tuple together with a boolean value that is `true` only when the user has finished choosing both endpoint nodes. This positive truth value provides indication to the other methods that the collapsing procedure is ready to proceed upon request by the user.

`method begin_collapse_selection : int -> unit`

This method is called when the user makes his or her selection of the first node that represents an endpoint node in a yet to be executed collapsing procedure.

`method end_collapse_selection : int -> unit`

Similarly to the `begin_collapse_selection` method described above, this method is called when the user selects the second node that represents an endpoint in a yet to be executed collapsing procedure.

`method show_children : int -> unit`

This method is used only for debugging purposes. It outputs in the command shell the names of the children of the node associated with the selection value provided by the argument.

```
val mutable temporary_uncollapsed :
  bool * String.t * String.t * float * float
```

This variable holds the information associated with the "temporary uncollapse procedure" as defined in the Design chapter.

```
method temporary_uncollapse : int -> unit
```

This method executes the temporary uncollapse procedure upon the user's request. To accomplish the procedure, the method relies on the information contained in the `temporary_uncollapsed` variable that is described above.

```
method collapse_back_the_temporary_uncollapsed : unit
```

This method voids the results achieved by the `temporary_uncollapse` method that is described above. In other words, it is used to restore the collapsing effect of the parts of the graph that were initially collapsed (before calling the temporary uncollapse procedure).

```
method uncollapse : int -> unit
```

This method permanently uncollapses a collapsed node. It completely restores the collapsed section of the deduction graph that was represented by the collapsed node associated with the supplied by the argument selection value.

```
method private collapse_procedure : String.t -> String.t -> unit
```

This method executes a collapsing procedure based on the preliminary selection of beginning and end nodes (see above). It not only instantiates a new object of the `Node` class and configures it, but also creates the new arrows connecting it to the other parts of the deduction graph, includes the collapsed parts of the graph in a list of objects to be made invisible, and deals with all peculiarities and algorithms mentioned in the collapsing section of the Design chapter.

```
val mutable collapse_history : (String.t * String.t) list
```

This variable contains a list tuples that represent the history of all collapsing procedures done so far in the process of exploring and manipulating the deduction graph in Panoptes. Each tuple holds two strings: the first string is the beginning node and the second string is the ending node for each collapsing procedure that was ever done. This way the comprehensive history of these operations is saved, so that when the user updates the graph with new incoming from IMPS information, Panoptes has a record of all collapsing procedures that need to be performed.

```
val mutable renaming_history : (String.t * String.t) list
```

Similarly to the above variable, this variable keeps a comprehensive history of all occasions of user induced renaming of nodes in the deduction graph.

```
method clear_history : unit
```

This method resets the history lists in the case of a global reset of Panoptes. Such events usually occur when the user starts developing a brand new proof in IMPS. Of course, this reset is requested by the user rather than happening automatically.

```
method execute_collapse : bool -> unit
```

This method invokes the collapsing procedure. The boolean value in the argument provides information to the system about whether the collapsing is a result of a direct request by the user, or it is an automatic history recall.

```
method rename_node : String.t -> String.t -> unit
```

This method is called when the user initiates a renaming procedure on a certain node. The string supplied by the first argument provides the old name (used by the method to identify the node to be renamed). The string supplied by the second argument contains the new name of the node.

```
method renaming_procedure : int -> unit
```

This method calls the `rename_node` method that is described above. It first identifies the node associated with the selection value provided by the argument.

```
val mutable dragged_node : bool * String.t * float *
  float * float
```

This variable holds a tuple of a few values. The boolean value is set to `true` if the user is dragging a node at the present moment, while the other values represent the name of the dragged node and its position coordinates at the moment the dragging started.

```
method mouse_left_button : int -> float -> float -> unit
```

This method is called each time the user presses down on the left mouse button. Based on the selection value and click location supplied by the arguments, the method calls all appropriate methods in order to process this event.

`method mouse_left_button_release : unit`

This method is called every time the user releases the left mouse button. Basically, the method clears all variables holding information regarding certain actions, such as the `locked_node`, the `locked_dg_drag`, and the `dragged_node` variables, all of which were described above.

`method mouse_left_button_dragging : float -> float -> unit`

This method is called when the user drags the mouse while holding down the left button.

`method mouse_right_button : int -> unit`

This method is called when the user presses the right mouse button over a certain object on the screen, which is identified by the selection value provided by the argument.

`method mouse_hover : int -> unit`

This method is obsolete at the moment, but it used to provide functionality for highlighting the nodes when the mouse cursor hovers over them. Subsequently, this feature of Panoptes was disabled as it severely degrades the performance of the system.

`method scan_for_hits : float -> float -> int`

This method utilizes the OpenGL functionality to send an imaginary ray from a certain location (supplied by the arguments) in order to obtain the object the ray hits (the ray travels perpendicularly to the computer screen, away from the user). The result is the selection value of the hit object, or a zero if the hit list is empty.

`method init : unit -> unit`

This method resets all class variables. It is also called during initialization when the class is first instantiated into an object.

`val mutable new_data_available : bool`

This is a variable that holds a boolean value. A `true` value indicates that the system should provide a notification message to the user regarding availability of new IMPS data, and prompt him or her to click the keystroke associated with the "update" function.

`method new_data_available_message : bool -> unit -> unit`

This method occasionally checks to determine if there is new IMPS data available.

`method draw_new_data_available_message : unit -> unit`

This method displays the message that notifies the user when new IMPS data is available.

`val mutable help_text : bool`

This variable indicates the status of the help screen. Its value is set to `true` if the help screen should be currently visible, and `false` otherwise.

`method toggle_help_text : unit`

This method toggle the status of the help screen between visible and invisible states.

`method display_help_text : unit`

This method displays the help screen.

`method draw : unit -> unit`

This method redraws the whole frame. It is called 30-60 times per second on average, which depends on how powerful the graphical card of the host machine is.

`method reshape : w:int -> h:int -> unit`

This method is called each time the Panoptes window is resized by the user. It contains functionality to be used for integrating additional functionality in Panoptes to control multiple screens independently (see Future Work chapter).

`method zoom_content_box : float -> int -> unit -> unit`

This method performs a zoom on a content box. The float number supplied by the argument is the chunk, by which the zoom should happen. It can be negative for zooming in or positive for zooming out. The integer number is the selection value of the content box to be zoomed on.

`method zoom : float -> int * int -> unit -> unit`

This method performs a zoom on the whole deduction graph by moving it further away or closer to the user in order to achieve zoom out or zoom in effect respectively.

```
method setOrthoProjection : unit
method resetPerspectiveProjection : unit
method bitmap_text : float -> float -> string -> unit
method renderBitmapString :
  float -> float -> float -> Glut.font_t -> string -> unit
```

These methods are used as a supplement by other methods for achieving different supporting operations.

```
end
```

This whole class can be considered as the main Panoptes engine. Most of the user induced operations are implemented in this class. In addition, the class is responsible for coordinating and invoking all drawing procedures used for visualizing all possible components of the deduction graph (nodes, arrows, paths, information boxes, collapsed boxes, help screen, new data notification, etc.).

B.9 Module Panoptes

`val testing : bool`

A variable of type boolean. A `true` value would run Panoptes in testing mode. This mode is used for obtaining input from test files instead of IMPS.

`val dg_filename : string`

The filename that contains the IMPS deduction graph.

`val sqn_filename : string`

The filename that contains the detailed information about the sequent nodes in the deduction graph.

`val grabber : Sqn_grabber.sqn_grabber Pervasives.ref`

An instance of the "sqn_grabber" class that is used for monitoring the changes occurring to the file that holds the detailed information of the sequent nodes.

`val dg : Parser.graph_data_class Pervasives.ref`

An instance of the "graph_data_class" class that is used for parsing the IMPS deduction graph file. The class also creates the internal data structure to hold the deduction graph information. Also, it collects the detailed information of all sequent nodes.

`val content_box_scroll_chunk : float Pervasives.ref`

A constant that is used to regulate the rate of scaling when using that operation on an information box.

`val scene_scroll_chunk : float Pervasives.ref`

A constant that is used to regulate the rate of zooming when using that operation on the graph.

`val main_GLUT : unit -> unit`

Initializes the OpenGL window and creates all keyboard and mouse event listeners. This is the main "loop" of Panoptes.

BIBLIOGRAPHY

[1] C. Bajaj, S. Khandelwal, J Moore, and V. Siddavanahalli. Interactive Symbolic Visualization of Semi-automatic Theorem Proving. Technical Report TR-03-37, University of Texas at Austin, 2003.

[2] E. Chailloux, P. Manoury, and B. Pagano. *Developing Applications With Objective Caml.* O'REILLY, Paris, France, 2000.

[3] Flying Frog Consultancy. C++ vs OCaml: Ray tracer comparison, 2007. Online at `http://www.ffconsultancy.com/languages/ray_tracer/comparison.html`.

[4] W. M. Farmer. What is formalized mathematics. Online at `http://www.cas.mcmaster.ca/~wmfarmer/CAS-734-06/slides/01-formalized-math.pdf`.

[5] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: System description. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 701–705. Springer-Verlag, 1992.

[6] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[7] W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS user's manual. Technical Report M-93B138, The MITRE Corporation, 1993.

[8] J. Garrigue. An Objective Caml interface to OpenGL, 2007. Online at `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgl.html`.

[9] Gold Standard Group and SGI. Survey Of OpenGL Font Technology, 2007. Online at `http://www.opengl.org/resources/features/fontsurvey/`.

[10] Gold Standard Group and SGI. OpenGL—The Industry Standard for High Performance Graphics, 2008. Online at `http://www.opengl.org`.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. pages 367–383, 2002.

[12] INRIA. The French National Institute for Research in Computer Science and Control, 2008. Online at `http://www.inria.fr/index.en.html`.

[13] INRIA. The Caml Language, 2008. Online at `http://caml.inria.fr`.

[14] D. Kapur and D. R. Musser. An overview of the Tecton proof system. *Theor. Comput. Sci.*, 133(2):307–339, 1994.

[15] M. Kaufmann and J Moore. Design goals of ACL. Technical Report 101, Computational Logic, Inc., August 1994.

[16] M. Kaufmann and J Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *Software Engineering*, 23(4):203–213, 1997.

[17] M. Kaufmann, J Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[18] X. Leroy, D. Doligez, K. Garrigue, D. Remy, and J. Vouillon. *The Objective Caml System: Documentation and User's Manual.* Institut National de Recherche en Informatique et en Automatique, France, May 16, 2007.

[19] K. Murphy. Why OCaml?, December 3, 2002. Online at `http://www.cs.ubc.ca/~murphyk/Software/Ocaml/why_ocaml.html`.

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

[21] D. L. Parnas. Designing software for ease of extension and contraction. In *Proceedings of the 3th international conference on Software engineering*, pages 264–277, 1978.

[22] D. L. Parnas and P. C. Clements. A rational design process: how and why to fake it. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT) on Formal Methods and Software, Vol.2: Colloquium on Software Engineering (CSE)*, pages 80–100, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[23] D. Rémy. Using, Understanding, and Unraveling the OCaml Language. In Gilles Barthe, editor, *Applied Semantics. Advanced Lectures. LNCS 2395.*, pages 413–537. Springer Verlag, 2002.

[24] AT&T Research. GraphViz–Graph Visualization Software, 2008. Online at `http://www.graphviz.org`.

[25] P. Rudnicki. An Overview of the Mizar Project. 1992.

[26] SGI. GLUT–The Opengl Utility Toolkit, 2008. Online at `http://www.opengl.org/resources/libraries/glut/`.

[27] D. Shreiner. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[28] Wikipedia. Argus Panoptes — Wikipedia, The Free Encyclopedia, 2007. Online (accessed 16-July-2007) at `http://en.wikipedia.org/w/index.php?title=Argus_Panoptes&oldid=144163854`.

[29] R. S. Wright, B. Lipchak, and N. Haemel. *OpenGL SuperBible (4th Edition).* Addison-Wesley, Boston, MA, USA, 2007.