Visual Occam: High level visualization and design of process networks

Mikolaj M. Slomka
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/thesesdissertations

Part of the Graphics and Human Computer Interfaces Commons, OS and Networks Commons, Programming Languages and Compilers Commons, and the Software Engineering Commons

Repository Citation
https://digitalscholarship.unlv.edu/thesesdissertations/875

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Theses, Dissertations, Professional Papers, and Capstones by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
VISUAL OCCAM: HIGH LEVEL VISUALIZATION AND DESIGN OF PROCESS NETWORKS

by

Mikołaj Michał Słomka

Bachelor of Science
University of Nevada, Las Vegas
2007

A thesis submitted in partial fulfillment of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2010
THE GRADUATE COLLEGE

We recommend that the thesis prepared under our supervision by

Mikołaj Michał Słomka

entitled

Visual Occam: High Level Visualization and Design of Process Networks

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Jan Pedersen, Committee Chair
Laxmi Gewali, Committee Member
Evangelos Yfantis, Committee Member
Aly Said, Graduate Faculty Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies and Dean of the Graduate College

August 2010
ABSTRACT

Visual Occam: High Level Visualization and Design of Process Networks

by

Mikołaj Michal Słomka

Dr. Jan Pedersen, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

With networks, multiprocessors, and multi-threaded systems becoming more common in our world it is increasingly evident that concurrent programming is not something to be ignored or marginalized even though many takes on concurrency (mainly by means of monitors or shared resources) have proven to be difficult to deal with on large scales. Thankfully, a good deal of work has already been done to combat this, through CSP, occam, and other such derivatives, to produce a scalable process oriented paradigm. Still, it is cumbersome to attempt to deal with the intricacies of such communicating networks down to every minutia; if, instead, it was possible to manage communicating elements on a higher level it would be far more practical to design large scale networks of processes!

As such, Visual Occam has been designed to automate some of the inner workings of occam to allow any user (novice or otherwise) the ability to create complex networks of communicating processes through easy to understand user interactions and interfaces. Taking a number of cues from digital circuit design software and modern
integrated development environments, it is possible to select components (both pre-
defined and arbitrarily complex user created systems) from a library of objects, hook
them together in a network, and produce compilable code without having to worry
about how or why the chosen components perform their function. Since any of these
components may themselves be networks of processes, it becomes trivial to construct
large systems that would otherwise be unwieldy to put together by hand.

The end result? A high level, easy to understand, visual abstraction of those
concurrent networks previously so frustrating to develop.
ACKNOWLEDGMENTS

With many thanks to Dr. Pedersen for his continued supervision and patience during this project, my mother for her unwavering support, and my brother for being exactly as he is.
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER 2 PROCESS ORIENTED DESIGN</td>
<td>3</td>
</tr>
<tr>
<td>CSP</td>
<td>3</td>
</tr>
<tr>
<td>Background</td>
<td>3</td>
</tr>
<tr>
<td>General Introduction</td>
<td>4</td>
</tr>
<tr>
<td>Verification</td>
<td>9</td>
</tr>
<tr>
<td>Process Oriented Programming</td>
<td>10</td>
</tr>
<tr>
<td>JCSP</td>
<td>10</td>
</tr>
<tr>
<td><strong>occam</strong></td>
<td>12</td>
</tr>
<tr>
<td>Syntax</td>
<td>12</td>
</tr>
<tr>
<td>Examples</td>
<td>21</td>
</tr>
<tr>
<td>Programming with Pictures</td>
<td>22</td>
</tr>
<tr>
<td>CHAPTER 3 UI DEVELOPMENT</td>
<td>24</td>
</tr>
<tr>
<td>Design Philosophy</td>
<td>24</td>
</tr>
<tr>
<td>Layers</td>
<td>25</td>
</tr>
<tr>
<td>Usable</td>
<td>28</td>
</tr>
<tr>
<td>Implementation Details</td>
<td>29</td>
</tr>
<tr>
<td>UML</td>
<td>29</td>
</tr>
<tr>
<td>Object Oriented</td>
<td>33</td>
</tr>
<tr>
<td>Event Driven</td>
<td>37</td>
</tr>
<tr>
<td>Usability</td>
<td>40</td>
</tr>
<tr>
<td>CHAPTER 4 VISUAL OCCAM</td>
<td>42</td>
</tr>
<tr>
<td>Terminology</td>
<td>42</td>
</tr>
<tr>
<td>General UI Overview</td>
<td>44</td>
</tr>
<tr>
<td>Menu and Tool Bars</td>
<td>44</td>
</tr>
<tr>
<td>Main Panel</td>
<td>46</td>
</tr>
<tr>
<td>Minimap Panel</td>
<td>47</td>
</tr>
<tr>
<td>Code Outliner</td>
<td>47</td>
</tr>
<tr>
<td>Code Component Library</td>
<td>48</td>
</tr>
<tr>
<td>Tools / Modes Palette</td>
<td>49</td>
</tr>
<tr>
<td>Property Panels</td>
<td>55</td>
</tr>
<tr>
<td>Output Tabs</td>
<td>58</td>
</tr>
<tr>
<td>Example Work Flow</td>
<td>59</td>
</tr>
<tr>
<td>CHAPTER 5 EXPERIENCES</td>
<td>67</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>----</td>
</tr>
<tr>
<td>Design Iterations</td>
<td>67</td>
</tr>
<tr>
<td>Conceptualization</td>
<td>67</td>
</tr>
<tr>
<td>Rapid Prototype</td>
<td>68</td>
</tr>
<tr>
<td>Iterative Development</td>
<td>68</td>
</tr>
<tr>
<td>Interface Usability</td>
<td>69</td>
</tr>
<tr>
<td>Anticipating Actions</td>
<td>69</td>
</tr>
<tr>
<td>User Orientation</td>
<td>70</td>
</tr>
<tr>
<td>Program Structure</td>
<td>72</td>
</tr>
<tr>
<td>Visual Occam Limitations</td>
<td>74</td>
</tr>
<tr>
<td>Swing</td>
<td>76</td>
</tr>
<tr>
<td>Versatility vs Power vs Expectations</td>
<td>77</td>
</tr>
<tr>
<td>Layout Managers</td>
<td>77</td>
</tr>
<tr>
<td>Non-Static Elements</td>
<td>78</td>
</tr>
<tr>
<td>SwingX</td>
<td>78</td>
</tr>
<tr>
<td>Auto Generated Code</td>
<td>79</td>
</tr>
<tr>
<td>CHAPTER 6 RELATED WORK</td>
<td>80</td>
</tr>
<tr>
<td>Visual Programming</td>
<td>80</td>
</tr>
<tr>
<td>CSP</td>
<td>80</td>
</tr>
<tr>
<td>occam</td>
<td>81</td>
</tr>
<tr>
<td>Circuit Design</td>
<td>81</td>
</tr>
<tr>
<td>Graphical Programming Environments</td>
<td>82</td>
</tr>
<tr>
<td>Design and Usability</td>
<td>83</td>
</tr>
<tr>
<td>Visualization Paradigms</td>
<td>84</td>
</tr>
<tr>
<td>Overviews</td>
<td>84</td>
</tr>
<tr>
<td>Multiple Window Interfaces</td>
<td>85</td>
</tr>
<tr>
<td>Dual Visualizations</td>
<td>85</td>
</tr>
<tr>
<td>Visual Occam</td>
<td>85</td>
</tr>
<tr>
<td>CHAPTER 7 CONCLUSION</td>
<td>87</td>
</tr>
<tr>
<td>CHAPTER 8 FUTURE WORK</td>
<td>89</td>
</tr>
<tr>
<td>APPENDIX A ALGORITHMS</td>
<td>91</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>92</td>
</tr>
<tr>
<td>VITA</td>
<td>98</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

3.1 Visual Occam source code layer hierarchy. ........................................ 26
3.2 Command Communication Design Pattern. ......................................... 27
3.3 Information Communication Design Pattern. ..................................... 27
3.4 Visual Occam Source Code Layer Communication. ............................... 28
3.5 Class A is associated with Class B. ............................................... 31
3.6 Class A is an aggregation of class B. ............................................. 31
3.7 Class A is composed of Class B. ................................................... 31
3.8 Class A generalizes Class B. ....................................................... 32
3.9 Class A is realized by Class B. ..................................................... 32
3.10 Interface / Inheritance diagram of displayable source-layer elements. .... 35
3.11 Information Hiding Design Pattern. ............................................. 36
3.12 Class diagram of major source-layer elements. ................................ 37
3.13 Major event handler design patterns for top level layer commands. ...... 39
3.14 Major event handler design patterns for lower level layer commands. ... 40

4.1 Visual Occam User Interface. ....................................................... 45
4.2 Visual Occam Menus and Tool Bars. .............................................. 46
4.3 Visual Occam Main Panel. ............................................................. 47
4.4 Visual Occam Minimap. ............................................................... 48
4.5 Visual Occam Code Outliner. ........................................................ 48
4.6 Visual Occam Code Component Library. ......................................... 49
4.7 Visual Occam Tools Palette. .......................................................... 50
4.8 Visual Occam Select / Move / Resize Icons. ..................................... 53
4.9 Visual Occam Source Based Procedure Properties Panel. ................... 57
4.10 Visual Occam Node Properties Panel. ........................................... 58
4.11 Visual Occam Channel Properties Panel. ....................................... 59
4.12 Visual Occam Output Tabs. ........................................................... 59
4.13 occam implementation of a 1:2 MUX gate. ..................................... 60
4.14 occam implementation of an AND gate. ......................................... 60
4.15 occam implementation of an OR gate. ............................................ 61
4.16 occam implementation of a NOT gate. ........................................... 61
4.17 occam implementation of a NAND gate. ......................................... 61
4.18 occam implementation of a XOR gate. ........................................... 62
4.19 occam network of a XOR gate. .................................................... 62
4.20 occam implementation of a single bit adder with carry. ...................... 64
4.21 occam network of a single bit adder with carry. .............................. 64
4.22 occam implementation of a four bit adder with carry. ....................... 65
4.23 occam network of a four bit adder with carry. ................................ 65
4.24 occam implementation of a full adder with carry. ........................... 66
4.25 occam network of a full adder with carry. .................................... 66

viii
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Object properties panel design conceptualization.</td>
<td>68</td>
</tr>
<tr>
<td>5.2 Object properties panel design prototype.</td>
<td>69</td>
</tr>
<tr>
<td>5.3 Object properties panel design iteration.</td>
<td>70</td>
</tr>
<tr>
<td>5.4 Object hierarchy visualized as a tree.</td>
<td>72</td>
</tr>
<tr>
<td>5.5 Object hierarchy visibilities issues.</td>
<td>73</td>
</tr>
<tr>
<td>5.6 Non-static user interface components.</td>
<td>78</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Over the past few decades, parallel computing has been clearly picking up mainstream headway: systems are not only becoming faster, but new architectures are implementing multiple on-die cores or bringing multiple processors into the realm of the everyday consumer. It is surprising, then, to realize that software support for such multitasking systems has fallen far behind and, indeed, it is perceived difficulties (either as a result of language support or preconceived notions) that hold wholesale development back. The argument can be made, however, that such difficulties are only artifacts of flawed methods for managing process networks and that by changing these very methods we may yet breathe new life into parallel programming on a truly massive scale.

Within this work we seek to discuss the development, from design philosophy to implementation details, of a tool that properly addresses many of these difficulties, allowing a user to quickly construct visual representations of complex networks and generate equivalent occam code at the touch of a button. By literally programming with pictures, the user gains the benefit of not only the solid mathematical background on which occam/CSP is based, but also of the condensed information diagrammatic environments provide. Furthermore, support is extended to allow easy reuse of code, duplication, manipulation, and all manner of other familiar techniques to uphold general claims of usability.

Speaking strictly organizationally, this thesis begins with an in depth introduction to the background concepts necessary to appreciate process oriented design, including the CSP model of parallel computing and general occam syntax. It continues, in Chapter 3, with design and implementation details for the Visual Occam application while Chapter 4 provides both a comprehensive overview of the interface and an example work flow that showcases the many strengths of visual programming. This is all followed by a discussion of some of the major problems and experiences encountered
during the course of the project in Chapter 5, a survey of related works in Chapter 6, and finally conclusions and potential avenues for future work in Chapters 7 and 8 respectively.
CHAPTER 2

PROCESS ORIENTED PROGRAMMING AND DESIGN

The world in which we live is a complex network of (largely) independent agents all acting and interacting with each other. So often are we exposed to such systems that it should be a simple matter to create a computational model that naturally reflects the interplay contained therein: though the patterns constructed may be complex, it is inevitably ingrained in the way we think and act. As noted by Welch [Wel02] however, concurrency is usually treated as an advanced topic that, while possible to understand, is difficult to implement. In a classic example, Java threads come packaged with a myriad of warnings regarding iconic issues (race hazards, deadlock, and livelock) and are certainly almost never recommended in an initial solution [OW04, Bel05a]. Indeed, this kind of behavior is quite the opposite of what should be expected from such a naturally occurring concept: concurrency needs to simplify implementations and design. As such, an entirely different model is instead considered.

2.1 CSP

Communicating Sequential Processes (CSP) [Hoa78, BHR84, Hoa85, RHB97] is, in the most basic sense, a style of notation borne of the need to accurately describe the interactions that may occur between concurrent agents. More specifically, it is a calculus that strives to formally model concurrent systems and provides the necessary tools to delineate the communication and synchronization contained therein. As with other calculi, CSP implements a collection of primitives, operators, and governing laws that allow concurrent systems to be discussed with some modicum of rigor.

2.1.1 Background

While the theories behind CSP were originally described by Hoare in 1978 [Hoa78], the focus was more on the description of a proto-language to solve a problems in concurrent programming rather than the construction of a rigorous algebra. To that end, it bore only passing resemblance [RHB97] to the later revised model [BHR84,
Hoa85]. It is this second version, and the subsequent modifications, that form the
basis for much of the current research in the specification and, more poignantly,
verification of concurrent systems.

2.1.2 General Introduction

Though it is outside the scope of this discussion to delve into every nuance of
CSP, no discourse on Process Oriented Design should exist without at least a crash
course. Adapted from Hoare’s originally publications on the matter [Hoa85], then, a
quick tour of CSP is provided.

2.1.2.1 General Notation

First, as is necessary when describing systems, the language to be used for the
remainder of the discussion is itemized. In particular:

- Words that are written in all lower-case letters are considered distinct events,
as are letters themselves: event, coin, a, c.

- Words that are written in all upper-case letters define specific processes: PROCESS, VMS. The letters P, Q, R are reserved for arbitrary processes.

- The letters x, y, z are variables that refer to events.

- The letters A, B, C are variables denoting a sets of events.

- The letters X, Y are variables that stand for processes.

2.1.2.2 Alphabets and Processes

All processes have an alphabet, denoted by $\alpha_P = \{event_1, \ldots, event_n\}$, which
is defined to be the set of names of all events that may be considered relevant to
the process itself. Given that no event may occur to or for a process without being
included in said process’s alphabet, $\alpha_P$ must necessarily describe all the possible
interactions with process $P$. It is important to note that there is no distinction made
between events that are initiated by the process itself and those that have origin in
an outside process. A simple vending machine, then, may have an alphabet similar
to this:
\[ \alpha_{VMS} = \{\text{coin, } \text{choc}\} \] (2.1)

While a more complicated vending machine (something that produces change for wrong denominations of coins used) would have to have an extended alphabet:
\[ \alpha_{VMC} = \{\text{coin}_{\text{small}}, \text{coin}_{\text{large}}, \text{return}_{\text{small}}, \text{choc}\} \] (2.2)

Special mention is made for a process with alphabet \( A \) that never utilizes any of the events \( x \in A \). Denoted as \( \text{STOP}_A \), this process describes a broken object, one that is capable of interacting but never explicitly engages in such events. It is further acknowledged that as a result of the given conventions, different alphabets have different \( \text{STOP} \) processes even though they behave (by not responding) in the same manner: \( \text{STOP}_{\alpha_{VMS}} \neq \text{STOP}_{\alpha_{VMC}} \).

2.1.2.3 Prefix and Recursion

A prefix, denoted by \( (x \rightarrow P) \), describes a process that first performs event \( x \) and then behaves as process \( P \). By definition, this prefix process must have the same alphabet as \( P \), or more formally: \( \alpha(x \rightarrow P) = \alpha P \). In general, prefix notation is used to describe the sequential flow of events in any given process. For example, it is possible to describe a simple vending machine using the alphabet previously described by 2.1 that takes in a coin and dispenses chocolate precisely one time:
\[ VMS = (\text{coin} \rightarrow (\text{choc} \rightarrow \text{STOP}_{\alpha_{VMS}})) \] (2.3)

Naturally, as it would not be possible to explicitly write out the entire event and process chain for a vending machine that serves an arbitrary number of customers, it is necessary for CSP to also include notation for repeated tasks. By use of simple recursion, we have:
\[ \mu X : A \bullet F(X) \] (2.4)

Where \( X \) is a locally bound variable, \( F(X) \) is a prefix process that contains the name \( X \), and \( \alpha X = \alpha F(X) = A \). We can thus define a perpetual vending machine.
(again using the VMS alphabet from 2.1) as follows:

\[ VMS = \mu X : \{ \text{coin, choc} \} \bullet (\text{coin} \rightarrow (\text{choc} \rightarrow X)) \] (2.5)

or with more simple syntax, by omitting the mention of the obvious alphabet used:

\[ VMS = (\text{coin} \rightarrow (\text{choc} \rightarrow VMS)) \] (2.6)

2.1.2.4 Choice

Powerful as recursion may be, it cannot properly encompass those processes that are influenced by outside agents, let alone those that perform different functions based on different input. To remedy this, the choice operator is introduced.

\[ (x \rightarrow P \mid y \rightarrow Q) \] (2.7)

This notation describes a process that performs either \( x \) or \( y \) and then behaves as \( P \) if the action performed was \( x \) or \( Q \) if it was \( y \) instead. If \( x \neq y \), it becomes clear that the process behavior (literally, the choice between \( P \) or \( Q \)) is determined by whichever event occurs first. Note that alphabet consistency is necessarily maintained:

\[ \alpha(x \rightarrow P \mid y \rightarrow Q) = \alpha P \text{ if } \{x, y\} \subseteq \alpha P \text{ and } \alpha P = \alpha Q. \]

Using this operator, it becomes possible to describe a vending machine that produces two different kinds of goods (at the discretion of the user) after accepting a coin:

\[ \alpha VMCT = \{ \text{coin, choc, toffee} \} \] (2.8)

\[ VMCT = \mu X \bullet (\text{coin} \rightarrow (\text{choc} \rightarrow X \mid \text{toffee} \rightarrow X)) \] (2.9)

2.1.2.5 Combining Processes

Of course, while simply describing the behavior of single, sequential machines may be somewhat interesting, it of greater importance to discuss how such processes interact with each other and in what ways they may be combined to form more complex systems.
Parallel Composition  Denoted by \(( P \parallel Q )\) where \(\alpha(P \parallel Q) = \alpha P \cup \alpha Q\), parallel composition describes a process that behaves like a system in which \(P\) and \(Q\) are allowed to run independently but may further interact with each other in a synchronized fashion. That is to say, events in the system may only occur when both of the composing processes are ready. Without loss of generality, if \((x \in \alpha P) \land (x \notin \alpha Q)\) then the event, \(x\), is of no operational concern to \(Q\) and is not considered. For example:

\[
\alpha GRDCUST = \{ coin, choc, toffee \} \tag{2.10}
\]

\[
GRDCUST = \mu X \bullet (toffee \rightarrow X \mid coin \rightarrow choc \rightarrow X) \tag{2.11}
\]

\[
( VMCT \parallel GRDCUST ) = \mu X \bullet (coin \rightarrow choc \rightarrow X) \tag{2.12}
\]

Shows that even though a greedy customer may want to get a toffee without first inserting a coin, as the vending machine system described previously (2.9) does not have that capability, the composition of these two processes has only one possible outcome.

Sequential Composition  Denoted by \(( P ; Q )\) where \(\alpha(P ; Q) = \alpha P \cup \alpha Q\), this describes a process that first behaves like \(P\), allows \(P\) to terminate, then behaves like \(Q\), allows \(Q\) to terminate, and finally behaves like \(\checkmark\) (the successfully completed process). It is important to note that, unlike parallel composition, both \(P\) and \(Q\) must also be sequential processes for this to make sense and thus both must necessarily have \(\checkmark\) in their alphabet. Such processes may be designed by including the \text{SKIP}_A\) process, which does nothing but terminate \((\alpha \text{SKIP}_A = A \cup \{ \checkmark \})\). Naturally then, it should be clear that if either \(P\) or \(Q\) never terminates than neither can their composition, \(( P ; Q )\). For example, it is possible to describe a vending machine that serves exactly two customers by combining multiple instances of a single serving vending machine:

\[
\alpha VM = \{ coin, choc, SKIP \} \tag{2.13}
\]

\[
VM_{ONE} = ( coin \rightarrow (choc \rightarrow SKIP \mid toffee \rightarrow SKIP)) \tag{2.14}
\]

\[
VM_{TWO} = ( VM_{ONE} ; VM_{ONE} ) \tag{2.15}
\]
We may also represent, recursively, the concatenation of arbitrarily many serial processes by the $*P$ notation where:

$$
*P = \mu X \cdot (P ; X)
$$

(2.16)

Thus, to get a vending machine that serves as many customers as is necessary (and functions identically to the recursively defined process 2.6), we state:

$$
VMS_{CST} = *VMS
$$

(2.17)

2.1.2.6 Nondeterminism

In contrast to the previously defined choice operators, nondeterminism allows for selection between processes with little or no influence from the outside world. We begin by introducing $(P \sqcap Q)$, where $\alpha(P \sqcap Q) = \alpha P = \alpha Q$, most easily conceptualized as a process that executes either $P$ or $Q$ in such a manner that selection between the two processes is completely arbitrary (that is to say, with no outside influence whatsoever). As an example,

$$
\alpha VMR = \{\text{coin, choc, toffee}\}
$$

(2.18)

$$
VMR = (\text{coin} \rightarrow ((\text{choc} \rightarrow VMR) \sqcap (\text{toffee} \rightarrow VMR)))
$$

(2.19)

will give either a choc or a toffee whenever a customer inserts a coin, but without providing the customer with a mechanism for choosing which. As this kind of decision is completely random, however, it does not completely describe ideal nondeterministic events; the controlling environment needs to be able to deal with both $P$ and $Q$ while never knowing which operates when. To combat this, a new all purpose notation is used that will, unlike $(P \sqcap Q)$, consider some amount of environmental influence: $(P \sqcup Q)$. Without loss of generality, if $(P \sqcup Q)$ and $P$ is not possible, then $Q$ will be engaged. Formally:

$$
((c \rightarrow P) \sqcup (d \rightarrow Q)) = ((c \rightarrow P) \mid (d \rightarrow Q)) \quad \text{if } c \neq d.
$$

(2.20)

$$
= (P \sqcap Q) \quad \text{if } c = d.
$$

(2.21)
2.1.7 Communication

Of course, no discourse on Communicating Sequential Processes would be complete without introducing the mechanisms necessary to describe the very interactions between processes that make large networks so interesting. Communication in CSP is an event described by a pair \( c.v \), with \( c \) denoting the name of the channel across which such communication occurs and \( v \) representing the value passed (formally, \( \text{channel}(c.v) = c \) and \( \text{message}(c.v) = v \)). Given this, we then construct the set of all messages which \( P \) can communicate on channel \( c \) as follows,

\[
\alpha_c(P) = \{ v \mid c.v \in \alpha_P \} \quad (2.22)
\]

and only allow communication on paired input (\( c!v \)) and output (\( c?v \)) processes, themselves following formally defined behavior:

\[
(c!v \rightarrow P) = (c.v \rightarrow P) 
\]

\[
(c?v \rightarrow P(x)) = (y : \{ y : \text{channel}(y) = c \} \rightarrow P(\text{message}(y))) \quad (2.24)
\]

On the condition that the alphabets of the channels that connect individual processes remain consistent: \( \alpha_c(P) = \alpha_c(Q) \). This, of course, allows for more direct specification of process behavior, as in the following bit copier.

\[
\alpha_i(\text{COPY}) = \alpha_o(\text{COPY}) = \{ 0, 1 \} \quad (2.25)
\]

\[
\text{COPY} = \mu X \bullet (i?x \rightarrow (o!x \rightarrow X)) \quad (2.26)
\]

2.1.3 Verification

As mentioned earlier, all of this serves to allow CSP to be used to speak about the networks defined in manners not previously envisioned—formal analysis and verification truly shows the extent to which CSP can be used. Though the mechanics behind such verification is typically reserved for powerful tools [For05] and far more in depth discussions are available directly from Hoare [Hoa85], Roscoe [RHB97], and others [BHR84], it is of some benefit to at least point out that such proofs can exist and even show how, at least in concept, they work. We define, then, three semantic
models and note that an entire set of operators, laws, and axioms is defined for each [Hoa85].

**Trace** The set of all sequences of events that a process may perform such that
\[
\text{trace}(\text{STOP}) = \{\langle \rangle \}.
\]
It should be clear that the trace of a large process network can get very large very quickly.

**Failure** Denoted as a pair, \((s, X)\), where \(s\) is a trace and \(X\) is the set of all processes that no longer function once \(s\) has been performed.

**Divergence** A finite trace during (or after) which a process falls to performing an infinitely long sequence of internal commands, thus failing to respond to external influences or commands. This is akin to a livelock situation.

Such models, and the careful combination of the laws for each, create the basis for verification as a whole. Making it possible to test CSP specifications for livelock, deadlock and, with refinement, even equivalence to other networks. Furthermore, such concepts (and the tools like FDR [For05] that implement them) allow holistic testing, checking of program correctness, and even rudimentary proof of the absence of bugs [Mit08, HBB+09].

### 2.2 Process Oriented Programming

Naturally, given the possibility of formal verification, it would be very interesting to be able to create algorithms (or even software) using the laid out methodologies. Unfortunately, CSP is itself simply a language used to describe such theories and does not have support for full fledged development. Thankfully though, work has already been done that addresses this glaring lack of support.

#### 2.2.1 JCSP

Welch’s JCSP [Wel02] seeks to extend the mathematical models of the CSP calculus into the practical programmer’s world. While there are certainly a number of libraries for Java that provide concurrent behavior (e.g. monitor-threads) [OW04], the inherent complexities of systems constructed using these models make it difficult
to reason about programs of even only moderate scale; the resulting interdependencies make race-hazards, deadlocks, livelocks, and other multi threading pitfalls far more troublesome to overcome. In turn, basing concurrency on the CSP model allows Java to inherit a rich set of provable properties, produce cleaner systems, and maintain confidence that formal verification exists all while benefiting from the original language design goals [GM96] that made Java what it is today [Dar04]. JCSP has thus been developed into library of primitives, extensions, and wrappers [Bel05b, Bel05c] enabling any programmer to implement multi threaded systems entirely in terms of CSP’s basic constructs without necessarily having to fully grasp the mathematics behind it.

Without spending too much time formally describing the system, the following example is provided without loss of generality:

```java
public class ExampleProcess implements CSProcess {

    public void run() {

        One2OneChannelInt a = new One2OneChannelInt();
        One2OneChannelInt b = new One2OneChannelInt();

        new Parallel {
            new CSProcess[] {
                new ProcessA(a, b),
                new ProcessB(b, a)
            }
        }.run();
    }
}
```

We see that this framework defines `ExampleProcess` as a CSP like process that behaves as the parallel composition of `ProcessA` and `ProcessB`. Since similar such
constructions necessarily exist for all the important CSP concepts, it is clear that
JCSP allows for the quick buildup of communicating networks with similar properties
to that of its ancestral namesake.

2.3 occam

In light of the difficulties presented in attempting to ratchet the formal concepts
derived from CSP into currently existing languages (especially those not necessarily
designed to handle concurrency in the manner dictated), it is somewhat beneficial to
instead consider those languages built around such models. occam, derived directly
from Hoare’s concepts [MT84, Way87], serves as one such realization, focusing on the
communication and lightweight concurrency specified as vital to CSP. Furthermore,
by starting on a fresh base and constructing with such support in mind, it is possible
for occam to avoid the dramatic verifications of the very systems one is attempting
to avoid [WM00]. A walkthrough of occam is thus provided as a backbone for further
discussion [MT84, New86, EH87, JG89, SGS95].

2.3.1 Syntax

In the most basic sense, occam code is conceptualized as a layered network of
processes each of which may itself be another network. A formal, but not exhaustive,
description of the occam grammar is provided in an Augmented-BNF [SGS95]:

\[
\langle \text{process} \rangle \rightarrow \langle \text{assignment} \rangle \mid \langle \text{input} \rangle \mid \langle \text{output} \rangle \mid \text{SKIP} \mid \text{STOP} \\
| \langle \text{sequence} \rangle \mid \langle \text{parallel} \rangle \\
| \langle \text{selection} \rangle \mid \langle \text{loop} \rangle \mid \langle \text{alternation} \rangle \\
| \langle \text{instance} \rangle \\
| \langle \text{specification} \rangle \\
\langle \text{process} \rangle
\]

2.3.1.1 Primitive Processes

All occam programs are considered networks of processes, even the most basic of
operations. Referred to as actions in occam literature, these primitive processes serve
as the cornerstone of basic development.

**Assignment** Simple assignment, which assigns some value to some variable is expressed as follows:

\[
\langle \text{assignment} \rangle \rightarrow \langle \text{variable} \rangle := \langle \text{expression} \rangle
\]

Where the resulting value of \( \langle \text{expression} \rangle \) must be of the same data type (see 2.3.1.2) as \( \langle \text{variable} \rangle \). It is further possible to perform multiple assignments by specifying variables and expressions as comma separated lists:

\[
\langle \text{assignment} \rangle \rightarrow \{1 , \langle \text{variable} \rangle \} := \{1 , \langle \text{expression} \rangle \}
\]

Here the expressions (on the RHS of the := operator) will be evaluated and then assigned to the corresponding variable from the LHS list. It is important to note that all such evaluations take place in parallel, as do all the assignments, resulting in a number of necessary extra rules governing placement and usage of names. In particular:

- No variable may appear twice in the assignment list.
- No variable that appears in the assignment list may be used in an expression that selects a component from an array.

**Communication** The communication action. The format and type of communication a channel is capable of transmitting is specified during declaration. Input is denoted by:

\[
\langle \text{input} \rangle \rightarrow \langle \text{channel} \rangle ? \langle \text{variable} \rangle
\]

which attempts to receive a value from the readable end of the specified channel and assign it to the given variable. This is necessarily paired with an output statement:

\[
\langle \text{output} \rangle \rightarrow \langle \text{channel} \rangle ! \langle \text{expression} \rangle
\]
which places the value of expression on the writable end of the specified channel then halts until the value is read by an input call to the same channel.

2.3.1.2 Data Types

As in other languages, occam values are split into different data types that determine not only an acceptable value range but what operations are considered valid.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>true or false</td>
</tr>
<tr>
<td>BYTE</td>
<td>[0, 255]</td>
</tr>
<tr>
<td>INT</td>
<td>variable(^1) twos compliment</td>
</tr>
<tr>
<td>INT16</td>
<td>([-32768, 32767]) twos compliment</td>
</tr>
<tr>
<td>INT32</td>
<td>([-2^{31}, 2^{31} - 1]) twos compliment</td>
</tr>
<tr>
<td>INT64</td>
<td>([-2^{63}, 2^{63} - 1]) twos compliment</td>
</tr>
<tr>
<td>REAL32</td>
<td>(\pm 2^{exp-127} \times 1.frac) (0 &lt; exp &lt; 255) (exp = 0) and (frac \neq 0)</td>
</tr>
<tr>
<td>REAL64</td>
<td>(\pm 2^{exp-1023} \times 1.frac) (0 &lt; exp &lt; 2047) (exp = 0) and (frac \neq 0)</td>
</tr>
</tbody>
</table>

Table 2.1: occam data types.

2.3.1.3 Replicator

Several occam elements may be replicated, producing a set of similar processes, analogous to the FOR-loop constructs available in other languages. It is necessary to provide an index, \(name\), which is initialized to a base integer value and incremented by 1 after each process execution until some \(count\) is reached. More formally:

\[
\langle\text{replicator}\rangle \rightarrow \langle\text{name}\rangle = \langle\text{base}\rangle \text{ FOR } \langle\text{count}\rangle
\]

\(^1\) Automatically resizes to the minimum word size (chosen from INT16-64) required to contain assigned values.
Where both \( \langle base \rangle \) and \( \langle count \rangle \) are either expressions or values.

It is important to note that the specified index may be referred to as an expression within the scope of the replicated process, but itself may not be the target of assignment. This allows the set of replicated processes to range across \([base, base + count]\).

For example, consider that:

```plaintext
SEQ i = 2 FOR 3
    channel ! i
```

Expands to:

```plaintext
SEQ
    channel ! 2
    channel ! 3
    channel ! 4
```

2.3.1.4 Sequence

As most basic of occam constructed processes, SEQ strings together processes for rigid sequential execution.

\[
\langle sequence \rangle \quad \rightarrow \quad SEQ
\]

\[
\{ \langle process \rangle \}
\]

\[
| \quad SEQ \langle replicator \rangle
\]

\[
\langle process \rangle
\]

Consider the following example:

```plaintext
SEQ
    keyboard ? char
    screen ! char
```

In this construction two distinct processes are combined: the first command receives a value from the \emph{keyboard} channel and assigns it to the \emph{char} variable, while
the second command then writes the previously received value to the \textit{screen} channel. As this particular set of operations only makes sense in the order specified, \texttt{SEQ} is ideal for enforcing the kind of sequential execution that is typically implicit in other languages.

2.3.1.5 Parallel

Similarly, \texttt{PAR} represents the parallel composition of processes:

\[
\langle \text{parallel} \rangle \rightarrow \text{PAR} \\
\quad \{ \langle \text{process} \rangle \} \\
\quad \text{PAR} \langle \text{replicator} \rangle \\
\quad \langle \text{process} \rangle
\]

Unlike \texttt{SEQ}, the order of execution here cannot be explicitly defined by the process. As such, it is important to notice that for processes like:

\begin{verbatim}
PAR
    process.1 ()
    process.2 ()
\end{verbatim}

the system may interleave \texttt{process.1} or \texttt{process.2} (see section 2.3.1.10 for details on specification and instantiation) in a different order each time the \texttt{PAR} process is executed. Because of this uncertainty, further stipulations are applied to both variable and channel assignment in \texttt{PAR} constructions to prevent potentially dangerous shared access situations. Specifically:

- No variable that is changed by input or assignment in one process may be read from in another.

- No channel may be used for input more than once nor for output more than once.

2.3.1.6 Construction

As expected from its relation to CSP, \texttt{occam} processes may also be built up from other process through a nested structure. This leads to formulations such as:
PAR
SEQ
  process.1 ()
  process.2 ()
SEQ
  process.3 ()
  process.4 ()

which starts both sequential processes at the same time and, without loss of generality, while the order of process.1 and process.2 is static, their operations may be interleaved with that of process.3 and process.4.

2.3.1.7 Conditionals

Standard conditional processes are also given to provide occam code with necessary flow control:

\[
\langle \text{conditional} \rangle \rightarrow \text{IF} \quad \{ \langle \text{choice} \rangle \} \\
\quad \mid \text{IF} \langle \text{replicator} \rangle \\
\quad \langle \text{choice} \rangle \\
\]

\[
\langle \text{choice} \rangle \rightarrow \langle \text{boolean} \rangle \\
\quad \langle \text{process} \rangle \\
\quad \mid \langle \text{conditional} \rangle \\
\]

In this construction, \(\langle \text{conditional} \rangle\) evaluates each \(\langle \text{boolean} \rangle\) expression in the defined order, stopping at whichever evaluates to TRUE first, executing the associated \(\langle \text{process} \rangle\), and finally terminating. As an example:

\[
\text{IF} \quad x < y \\
\quad x := 1
\]
This process necessarily sets the value of $x$ depending on its original relative value to $y$ but may not reach the final <conditional> expression to set $x := 0$. This results in $x \in \{1, 2, 3\}$ after execution.

### 2.3.1.8 Loop

Much like conditional processes, loops are provided in occam to provide further flow refinement:

$$\langle loop \rangle \rightarrow \text{WHILE } \langle boolean \rangle$$

$$\langle process \rangle$$

Here <process> is repeatedly executed until the <boolean> expression evaluates to FALSE. If <boolean> is initially FALSE, <process> will never be performed and entire loop will terminate. To illustrate:

```
WHILE x >= 0
  SEQ
    screen ! x
    x := x - 1
```

This process will repeatedly output the value of $x$ to the screen channel and then decrement the value, until the value itself is less than zero, resulting in output somewhat similar to $\{x, x - 1, \ldots, 1, 0\}$. Naturally, if $x$ begins less than zero nothing will be written to the screen channel.
2.3.1.9 Alteration

Alteration is a powerful concept in the occam world allowing the system to choose which process to execute based on some external factor.

\[
\langle \text{alteration} \rangle \rightarrow \text{ALT} \\
\{ \langle \text{alternative} \rangle \} \\
| \text{ALT} \langle \text{replicator} \rangle \\
\langle \text{alternative} \rangle
\]

\[
\langle \text{alternative} \rangle \rightarrow \langle \text{alternation} \rangle \\
| \langle \text{guard} \rangle \\
\langle \text{process} \rangle
\]

In this construction, each \langle process \rangle is guarded by a \langle guard \rangle which may or may not be ready. The ALT process, then, chooses whichever \langle guard \rangle is ready first, executes it, starts the associated \langle process \rangle and eventually terminates. Naturally, then, if none of the \langle guards \rangle are ready, the ALT process cannot continue. Conversely, if multiple \langle guards \rangle are ready, a single one is chosen nondeterministically—only one \langle guard \rangle and \langle process \rangle pair is ever handled by a single ALT. For example:

\[
\text{ALT} \\
\text{left} ? \text{packet} \\
\text{stream} ! \text{packet} \\
\text{right} ? \text{packet} \\
\text{stream} ! \text{packet}
\]

Writes information from the left or right channels to the stream, depending on whichever has information currently ready. If both left and right are ready to communicate when the ALT process starts, there is no way to predict which will be written to the stream channel.
2.3.1.10 Specification and Instantiation

Of final, but important, note is occam’s support for naming and instantiating processes. Dubbed procedures in the literature, these are used in much the same way procedures function in other languages.

\[
\langle declaration \rangle \rightarrow \langle type \rangle \{1, \langle name \rangle \} : \\
\langle definition \rangle \rightarrow \text{PROC } \langle name \rangle (\{0, \langle formal \rangle \})\\n\langle process \rangle : \\
\langle specification \rangle \rightarrow \langle declaration \rangle | \langle definition \rangle
\]

With the definition construction, it is clear that procedures necessarily take a minimum of three lines to specify. The first is the keyword PROC, followed by a name (to be used for later instantiation), followed by a list of zero or more formal parameters. The second is the process which, as per 2.27, may itself be made up of a series of specifications (see below) followed by a single process. The third is a simple : that denotes the end of the process definition.

\[
\langle specification \rangle \rightarrow \langle declaration \rangle | \langle definition \rangle
\]

Further, declaration of variables for use in the scope of their contained procedures is handled by first naming the type of variable followed by a list of at least one name to be used and terminated with a colon.

\[
\langle instance \rangle \rightarrow \langle name \rangle (\{0, \langle actual \rangle \})
\]

Finally, the actual instantiation of the procedure is handled by using the declared name followed by a list of actual parameters that correspond to the declared formal parameter list. The end result, of course, looks something similar to this:

\[
\text{PROC adder (CHAN INT a?, CHAN INT b?, CHAN INT c!)}\\
\text{INT x:}\\
\text{INT y:}
\]
SEQ
  a ? x
  b ? y
  c ! x + y

Which may be seen as a procedure that takes in two integers on two separate channels and outputs their sum to a third.

2.3.2 Examples

Putting all of this together, it becomes a simple matter to create large networks of processes and procedures:

PROC split (CHAN INT in?, CHAN INT out.a!, CHAN INT out.b!)
  INT x:
  WHILE TRUE
    SEQ
      in ? x
    PAR
      out.a ! x
      out.b ! x

PROC increment (CHAN INT in?, CHAN INT out!)
  INT x:
  SEQ
    out ! 0
  WHILE TRUE
    SEQ
      in ? x
      out ! x + 1

PROC numbers (CHAN INT out!)
CHAN INT a:
CHAN INT b:
WHILE TRUE

PAR
    increment (a?, b!)
    split (b?, a!, out!)

Where the procedure numbers outputs an ever incrementing stream of integers to the out channel by constructing a network that connects increment and split.

2.4 Programming with Pictures

As evident by even the smallest of examples, it is not only easy to get lost in the interconnections that occam (and thus CSP) networks inherently construct, but it is only natural to attempt to visualize these connections by means of some kind of graphical notation [Dow85, TTW97, JJS08]. Furthermore, it is abundantly clear that the processes used to construct networks in these methodologies operate independently of each other; the operational workings of a process is self contained outside of the occasional need for additional input and, indeed, there should be no reason why processes themselves cannot be completely blackboxed. As the user does not need to be aware of how a particular process works, but rather that it responds to communications requests as expected, dependence on visible code diminishes quickly and it thus makes sense to completely abstract the concept of processes and channels to a visual paradigm.

What is needed, it seems, is a development environment that allows a user to easily represent segments of code (say, an occam style procedure) and the connections these make to form a complete process network. Instead of coding, it should be possible for a user to quickly choose from a series of known processes, load everything up in some kind of view, connect accordingly, and compile. This kind of methodology, together with the ability to quickly reuse previously defined processes, would allow for the
construction of truly complicated systems without ever necessarily having to worry about the mechanisms behind them.
UI DEVELOPMENT

Of course before any development may begin in earnest it is beneficial to construct a concrete plan of action, otherwise the resulting work becomes unorganized and coding ineffectual; as shown time and time again [Kru03], it is surprisingly easy to complicate matters with poor choices early in the development cycle, doubly so when given the somewhat complex considerations involved in constructing user interfaces such as those proposed for Visual Occam. As such, a philosophy is defined to not only more properly guide all development efforts but also as a way to introduce discourse regarding numerous key elements that have clear ramifications on the final product.

Although the complete development of Visual Occam follows a rendition of Agile Programming (see section 5.1.3), it would, of course, be impractical to cover the intricacies of each iteration or lifecycle phase within the confines of this discussion. Instead, a detailed analysis of the resulting model is provided, starting from conceptual philosophies and culminating in a detailed description of implementation patterns derived, all of which sculpted the growth of Visual Occam.

3.1 Design Philosophy

Within the confines of the author’s experience, a design philosophy is a pivotal stepping stone in the construction of truly useful code: the axioms derived therein help reason around the potentially difficult choices to be made and help guide the evolution of the program as a whole. Such a stance is not much different from the rigorously developed guidelines for software engineering, be it the Unified Process [Kru03, Sta03], Agile Programming [HVZB04, WC03], or even the now outdated Waterfall Method [Lot97, Roy87, LB03]; generally speaking, having a plan of action is certainly better than having nothing at all. Of course, given the complex nature of the intended Visual Occam program, it would be fallacious to attempt to quantize the whole of the project without at least considering the individual components that make
it work. In that light, we start by breaking down Visual Occam into the constituent levels, describe the objects and interactions therein, and then finally attempt to piece it all together in under a single paradigm.

3.1.1 Layers

Visual Occam, as many graphical user interfaces, may be easily broken up into four hierarchically arranged layers (Figure 3.1):

**User** Defined in the most basic sense as the means through which a user interacts with the program itself, the *user*-layer displays all pertinent information and captures actions to be sent down to lower layers for analysis or processing. This layer is mostly generated by use of the integrated development platform’s (see section 6.2) graphical constructs and ties directly into both Swing and the Visual Occam’s specifically developed controls (see section 5.5).

**Interface** The *interface*-layer consists of those components and objects that serve as intermediaries between the *user*-layer and the Visual Occam source objects (literally, the *source*-layer). Treated mainly as a communications front, this layer helps ensure that the user interface itself may be constructed completely disjoint from the controlling Visual Occam source. Most of the Visual Occam meta-concepts, including libraries and collections, are handled here, along with the translation or breakup of *user*-layer events into lower level methods.

**Source** The vast majority of Visual Occam code is placed in the *source*-layer, including the structures that themselves represent the to-be-generated Occam source (*target*-code). These *source*-layer constructs handle most of the direct modification of such code by utilizing a number of *back-end*-layer functions and algorithms, but only allows communication or access through very specific commands.

**Back-end** All of the labor intensive or repeated calculations, along with all those methods that cannot otherwise be categorized, fall to this lowest layer. Containing mostly abstract concepts and algorithms, this layer is intended to be
freely called when necessary but should only be invoked by the source-layer itself to preserve integrity.

Given the desire to maintain a rigid code structure such as this, it becomes necessary to restrict the way in which communications are sent; should communications be formed freely between each layer the entire hierarchy would break down. We thus strive to separate communication into two categories:

**Command** Those communications or method calls that seek to modify some level of the target-code are designated as commands. No distinction is be made between those commands that successfully change the target-code and those that do not; success of the operation is not intrinsically tied to the communication. See Figure 3.2.

**Informative** Communications used for determining the state of any given object or component are considered informative. While the information returned by such communications may have some impact on the user interface, these changes are not explicitly a direct result of the calls themselves: in no way do informative communications modify the target-code or the state of the program itself. See Figure 3.3.
public class A {
    private B _target;
    private Command _c;
    public class B {
        private void source() {
            _target.command(_c);
        }
    }
}

(a) Source Object.  (b) Target Object.

Figure 3.2: Command Communication Design Pattern.

public class A {
    private B _target;
    private void source() {
        _target.get();
    }
}

(a) Source Object.

Figure 3.3: Information Communication Design Pattern.
Furthermore, it is important to note that all signals are defined with respect to the information itself; *command* communications operate by sending information to the called method’s object (originating at the caller) while *informative* communications send information back from the called method’s object and are referred to as originating at the callee. Using this vernacular, we may thus rigidly define communication pathways as seen in Figure 3.4.

Specifically, *command* communications may only ever be sent to a lower layer while *informative* communications may only ever be sent up to a higher one. By applying this formulation, we can easily see that command structure necessarily flows down while state information is always passed up through the hierarchy. More notably, at no point may an object in one layer request information from an object in a higher layer (and thus request that information be sent down), nor may such an object send commands to alter the target-code up to a higher layer. Further still, by restricting Visual Occam to such communications, we can safely construct layers in a disjoint fashion; the internal operations of any given layer are not necessary so long as they responded when prompted and within the expected operational constraints.

3.1.2 Usable

Aside from general functional and structural requirements, several usability concerns are raised as further major design points to be tackled directly during imple-
mentation. Specifically adapted from a number of works relating to the ISO standards on usability [LCM+04], we specifically want to ensure that Visual Occam falls well within the range of all four major metrics:

**Understandable** Software and interface functions must be easy to understand. In the general sense, the implication here is that all actions should be logically grouped and perform only what should be expected (an add function should not also delete something).

**Learnable** It should take minimal effort to learn the use of the application as a whole. Visual Occam attempts to hurdle this concept by maintaining roots in familiar settings. Concepts from other, more widely used programs, are applied wherever possible.

**Operable** The software is designed to be easy to use. Not only should it be transparent to the user what the flow of operations are, but those functions should be easy to access. Efficiency, in particular, is key.

**Attractive** In order for Visual Occam to stand against other platforms of the sort, general aesthetics need to be taken into account. Though general application style is outside the scope of what can be handled without delving deep into Java’s look-and-feel systems, layout, colors and displayed information may be tweaked to attempt to ensure a pleasing experience.

3.2 Implementation Details

With the design philosophies out of the way, it becomes possible to describe the model developed during the iterative process. We begin with a short introduction to the digram

3.2.1 UML

Developed as a general modeling system, the Unified Modeling Language [RJB04] is an industry standard method of visualizing and documenting the various engineering artifacts that come as a result of the development process. For the sake of
brevity, three major structures are used in all of Visual Occam’s design visualizations are described:

**Class Diagram** a UML diagram that describes the structure of a system by detailing classes, their attributes, and the relationships between each. For large or complex systems these are typically constructed for individual views that only represent portions of the whole.

**Class** A class to be realized in code. Visualized by a box with three sections: name, field(s), and method(s). Abstract or interface type classes are further denoted here as well.

**Relationship** Simply put, a programmatic connection between two classes. Represented by a single line with optional arrows, shapes, numbers, and headings depending on the type of relationship described.

Together with annotations and variance in connection type or line color (see Table 3.1), UML allows a designer to quickly specify the wide number of behaviors and structural elements found in code. It is primarily using this notation that the Visual Occam application model may be discussed.

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Line</th>
<th>Start</th>
<th>End</th>
<th>Figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation</td>
<td>Solid</td>
<td>None</td>
<td>None</td>
<td>3.5</td>
</tr>
<tr>
<td>Aggregate</td>
<td>Solid</td>
<td>None</td>
<td>Open Diamond</td>
<td>3.6</td>
</tr>
<tr>
<td>Composition</td>
<td>Solid</td>
<td>None</td>
<td>Shaded Diamond</td>
<td>3.7</td>
</tr>
<tr>
<td>Generalization</td>
<td>Solid</td>
<td>Arrow</td>
<td>Open Triangle</td>
<td>3.8</td>
</tr>
<tr>
<td>Realization</td>
<td>Dotted</td>
<td>Arrow</td>
<td>Open Triangle</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Table 3.1: Visualization subtypes.
Figure 3.5: Class A is associated with Class B.

Figure 3.6: Class A is an aggregation of class B.

Figure 3.7: Class A is composed of Class B.
Figure 3.8: Class A generalizes Class B.

Figure 3.9: Class A is realized by Class B.
3.2.2 Object Oriented

As should be obvious to the casual designer, speaking only in terms of layers and communications does not a program make; the individual components that any given layer or communication flow utilizes must be given some amount of attention, as there is both an intrinsic need to store the generated code or the user interactable details in some kind of logical manner and a desire for the those objects to be able to work together fluidly. Moreover, by working in a smaller resolution, it becomes possible to apply well documented patterns to further enhance design efforts. As such, we begin with layers.

Given that most of the concepts contained within a layer may be most easily described as concrete things, it is best to classify layers as sets of objects, be it buttons, menus, or even to-be-generated occam code. As of direct consequence of this classification, it becomes possible to apply a number of object oriented features to these components:

**Coupling**  As a measure of class interdependence or coordination, it is generally accepted that loose coupling is fundamentally indicative of good design. While in the ideal sense all classes and objects would only be able to communicate by passing parameter free messages across predetermined channels (not much unlike occam outright), such design is not necessarily practical given the tools and design patterns used. Therefore, it is instead necessary to attempt to minimize the amount of interdependence any given object has with another. In Visual Occam, no instantiated object shall have direct access or control to the inner workings of another such object, the two are only capable of interacting through either’s publicly available method calls. Small deviations from this rule are allowed for those abstract objects and interfaces used to construct much of object hierarchy, but this should have no visible entanglements to casual observers. This behavior clearly reflects the concepts outlined in the layered design philosophy: much as layers are loosely coupled, so should their inner workings.
**Cohesion** While often discussed alongside coupling, cohesion is itself a metric for shared responsibility or organization across all the members of a given class. Typically, cohesion increases the more related the member functions become, and as such, it is considered beneficial to have high degrees of cohesion. To ensure at least a modicum of such cohesion, it is simple enough to ensure that all members of a class help perform only those tasks that are necessarily related to the containing object (literally: functional cohesion). Again, this reflects directly on the idea that layers themselves are arranged into somewhat functional groups: individual layers need to be highly cohesive alongside their encapsulated objects. In terms of any layer’s responsibility, \( R(x) \), and the set of all objects, \( O \), this may be formally defined by \( \text{layer}_{R(x)} = \{ p, q \in O \mid R(p) \approx R(q) \approx R(x) \} \). Of course, as good metrics for responsibility are difficult to come by, a “good-enough” attitude is sufficient.

**Interfaces and Polymorphism** In line with the need to keep things loosely coupled, use of inheritance for object design is encouraged by means of interfaces where applicable. Acting as an abstract class type, interfaces in the Java sense may only contain references to potential methods or behaviors. It is through these interfaces that appropriate design patterns for message calling may be implemented and any object that later implements an interface must necessarily implement the specified methods. Furthermore, given that there are a multitude of objects within the Visual Occam source, it could be potentially advantageous to be able to speak about the more generic handles interfaces provide instead of the individually realized classes.

For example, both Procedures and Channels may be Visual Occam classes that share a need to be displayed by the user interface. While it is certainly necessary to worry about the specific implementations of these classes (how they are displayed), allowing both to implement a singular interface gives the designer a quick mechanic to refer to either object without necessarily specifying which. We can thus create a *displayable*-interface, with references methods to properly draw onto any given canvas, tie both Procedure and Channel directly into it as seen in Figure 3.10, and...
be confident that any container of Displayable objects will be capable of calling a display method on all members.

**Information Hiding**  Of course, given the scale of the Visual Occam project, ensuring loose coupling is by no means a trivial task. Extra measures, in the form of classic information hiding and wrapper functions are implemented to further protect individual classes from potential design mishaps and coding changes: all members of a class are defaulted to a private state (wherein no other class may have access) and publicly exposed methods that return copies of the requested information are provided where necessary (See Figure 3.11).

**Composition and Aggregation**  Unfortunately, the method by which Java handles object passing [AGH05] (specifically, object references as in Figure 3.11.(c)) complicates matters such as information hiding and coupling. Though Java itself passes information by value, object variables when instantiated are technically pointers. As a result of this, the passed values are themselves simply copies of pointers and while
public class A {
    private int _b;
    private C _c;
    ...
    ...
}

(a) Containing Class.  
(b) Primitive Wrappers.  
(c) Object Wrappers.

Figure 3.11: Information Hiding Design Pattern.

modifications to the pointer value will not result in changes elsewhere, usage of any of the object’s methods may. Thankfully, it is perfectly possible to design around this by conceptualizing such references as aggregates or compositions:

**Composition** Defined as a ‘has-a’ relationship, composed objects are those that exist as a collection of other objects. It is important to note that the object that is composed from other members is directly responsible for the existence of these members: removing the encapsulating object necessarily removes the composed items.

**Aggregate** The general case of the composition concept. Aggregation does not imply any kind of lifetime association between objects referenced; aggregates may be removed without destroying their encapsulated objects.

For example, though nodes are fully realized objects, it is is conceptually impossible for them to exist outside the bounds of an encapsulating procedure. Further still, collection procedures (as a concrete subtype of the general procedure concept) exist only as a set of procedures and associated channels, each of which is in turn created or dismantled by the encapsulating collection.

Given all of this, the daunting task of visualizing the general code structure becomes a bit simpler: we want to create a loosely coupled but highly cohesive set of objects by utilizing aspects of inheritance, information hiding, and aggregation
composition. Though the complete class diagram is far outside the scope of this discourse, Figure 3.12 clearly shows many such influences.

3.2.3 Event Driven

If objects are to be the realization of layers, it is clear then that events must relate somehow to the commands linking the source together. This is only natural, of course, as a collection of objects is outright useless without some way of method of interaction - double true given the program is expected to deal with potential user control. As such, we consider events, their handlers, and other such abstractions using the well documented tools of event driven programming; unorganized events, after all, are just as dangerous as poorly designed objects. We begin by classifying all events and handlers into the major categories previously associated with layer communication:

**Application** The main threads of operation from which all other work necessarily spawns. Only formally noted for completeness, this is essentially a program-
matic loop that awaits instruction or interruption by any of the further specified events and handlers.

**User** Those events whose direct ramifications are completely contained within the user interface. Button, keyboard, menu, and mouse presses (amongst others) initiated by the user all necessarily trigger such events and handlers. While a number of these are aesthetic handling events, many may trigger lower level handlers (specifically those in the interface-layer) to actually send commands to (or request information from) the source. As seen in Figure 3.13.(a), these handlers are in line with the layered code communication structure and clearly the means through which the user interacts with the user-layer but may not necessarily result in any modifications.

**Interface** Major application-source control is associated with the interface events and handlers. These events are those defined as either intermediaries between the user interface and the target-code or those that operate on meta-concepts such as libraries or projects. As noted in the layered philosophy discussion, the interface events exist primarily as a way to separate the IDE generated handlers from those that are hand crafted for Visual Occam (See Figure 3.13.(b)). At least one of these commands necessarily instantiates the source-layer for later control.

**Source** Direct modification of target-source and lower level systems is handled by events in the source-level. This includes, but is not limited to, those commands that instantiate new source-level objects, those that pass references from one source-level object to another, and those that help maintain the integrity of the target-source as a whole. Many of these event handlers use the back-end algorithms to perform their function. Adherence to the layer-communication pattern may be seen in Figure 3.14.(a), without loss of generally with regards to communication type (command or informative).

**Back-End** In the general sense, all the event handlers here are nothing more than
public Class UserInterface
{
    private Interface _i;

    public void handler(Event e)
    {
        /* UI Specific Commands */
        _i.handler(e);
    }
}

public Class Interface
{
    private Source _s;

    public void handler(Event e)
    {
        switch(e)
        {
            case A:
                _s.A(); break;
            case B:
                _s.B(Data d); break;
            ...
            default:
        }
    }
}

(a) User level.  (b) Interface level.

Figure 3.13: Major event handler design patterns for top level layer commands.

static methods and algorithms that are frequently used by the upper level layers
(Figure 3.14.(b)).

We can then see, at least from the this implementation specific perspective, that
relationships established for communication channels are trivially replicated in re-
strictions for event handling. It is of note, however, that in an attempt to save on
implementation complexity, methods (or events, in this case) themselves will only ever
call top-down; the upper states have to request state information regardless of what
changes may be occurring below. Thankfully, as communication events are defined
with respect to the flow of information, the integrity of the initial design philosophy
is maintained.

Taking this final point to a logical extreme, it should be clear that the entirety of
Visual Occam is constructed to be layered peers of objects that may only communicate
between layers when the user initiates some kind of action. Ideally, this is a reflection
on the methodologies of CSP itself - loosely coupled, virtually independent objects
Figure 3.14: Major event handler design patterns for lower level layer commands.

(procedures) interacting across some means of communication. The key result is, of course, that it becomes possible to blackbox the various layers in Visual Occam entirely and perform changes to portions of the code without necessarily effecting any other key components.

3.2.4 Usability

As with the general design philosophy, it is important to discuss how usability is addressed in the Visual Occam implementation design. After all, despite the fact that the user is far removed from the general details of a layered, event-driven application, it does little good to construct a system that is simply not usable. With Section 3.1.2 in mind, we have:

**Effective** To be considered *effective*, a system must first be both *understandable* and *learnable*. Thankfully, it is possible to argue that such an interface naturally stems from clean implementation: as each object or communication is grouped together with those that have similar responsibilities, it is necessarily difficult to construct an action that somehow combines conflicting responsibilities (it contradicts design to mix such groups). As such, so long as Visual Occam’s
implementation stays true to the discussed implementation patterns, it should remain effective. Furthermore, simple prompts and proper retrieval / display of state information (as is readily handled by communication protocols) further reinforces this metric.

**Efficient** It would be foolish to consider any application operable in a vacuum. Users will, invariably, use other tools to meet their goals and shape their expectations of the Visual Occam system. Taking advantage of this, it is possible to expand the efficiency a user may perform actions with by drawing influence from widely used applications. The interface design of Visual Occam, then, must insure high degrees of familiarity by careful analysis and mimicking of existing development environments (see section 6.2). Through such endeavors, it should be possible to introduce the best features taken from all into one package and thus enhance the speed at which even the most novice of users operates. Moreover, it is possible to apply some of the many features discussed in the volumes of literature dedicated to the topic, allocating time or work saving concepts into the design itself.

**Attractive** Although the most subjective of the usability concepts, by performing a number of design iterations on the interface itself, Visual Occam assures that at least some focus has been put forth towards aesthetic appeal.

As a final note, while it is clear that users, be it experienced developers or naïve programmers all should be able to approach the application with at least some success, the complete use of the metrics, tests, and refinements for usability study is considered outside the scope of the current Visual Occam project. The only reasonable way to measure how any of these usability paradigms are developed in Visual Occam is through general use testing, and since such testing has proved to be both difficult and, of course, time consuming it is reserved for future work (see section 8).
CHAPTER 4

VISUAL OCCAM

Although the discourse has been, thus far, mainly focused on the background of the Visual Occam platform, it cannot be forgotten that the eventual goal of all of this is to be able to program with pictures (see section 2.4). As is obvious by the amount of time spent in design, however, the construction of what is essentially a new integrated development environment is a non-trivial task, nor is the resulting application. While the entire system has been designed from the ground up to be user friendly (see section 3.1.2), it is still easy to get lost in the number of available features. Thus, a walkthrough of Visual Occam’s terminology, user environment, and code implementation / structure is offered along with a brief example of how all of this comes together to help the user construct viable occam code by simply clicking and dragging.

4.1 Terminology

Though a good majority of the terms and concepts discussed may be considered commonplace, certain vernacular choices have been made that require a bit of explanation before the discussion as a whole may begin. Formal models of the working spaces are also provided for reference:

Object An element or segment of code that has some kind of visual representation in the user interface. Notably: procedures, nodes, channels, and groups.

Procedure Objects that describe occam processes and function much in the same way procedures do in most other languages. They may contain source, some set of nodes, and all the elements or properties required to help eventually generate proper code.

Node Objects that describe the point of entry for procedure communication by means of some channel. These are the visual analogs to channel parameters
in oc cam.

**Channel** Objects that describe the path of procedure communication and function in much the same way channel variables do in oc cam.

**Source** Plain text oc cam code.

**Parameter** Additional information passed to a procedure / process during instantiation.

**Multiplex** The method by which Visual Occam describes a process instantiated multiple times under a par-for loop.

**Multi-Channel** A $n : m$ channel; a shared channel with multiple inputs or outputs.

**Group** A set of procedures, channels, and other groups.

**Bounds** Description of physical presence in any given space $S$: location $(x, y)$ and size $\{w, h\}$ such that $\{(x, y), (x + w, y + h)\} \in S$.

**monitor-space** The set of all double-precision values ($\mathbb{D}$) that may be contained within the current resolution:

$$M \equiv \{(x, y) | x \in \mathbb{D} \land y \in \mathbb{D}, 0 \leq x \leq Res_x, 0 \leq y \leq Res_y\}$$

where $(Res_x, Res_y)$ describes the current resolution. All valid mouse operations are necessarily contained within this space.

**work-space** The set that describe the current space usable for objects:

$$W \equiv \{(x, y) | x \in \mathbb{D} \land y \in \mathbb{D}, 0 \leq x \leq x_{max}, 0 \leq y \leq y_{max}\}$$

with $x_{max}$ and $y_{max}$ defined to be the minimum values required to encompass all objects contained within the space. Used when describing the bounds of an object or otherwise interacting with visual representations.
**scale-space** A transformation of *work*-space by some scalar value $\Delta$:

$$W_\Delta \equiv \{(f(x), f(y)) | x \in D \land y \in D, f : D \to D, f(x) = \lfloor \Delta x \rfloor\}$$  \hspace{1cm} (4.3)

where $\Delta_{\text{min}} < \Delta < \Delta_{\text{max}}$ and $\{\Delta_{\text{min}}, \Delta, \Delta_{\text{max}}\} \in D$.

**view-space** A subset of *scale*-space, bounded by $[(x_{\text{min}}, y_{\text{min}}), (x_{\text{max}}, y_{\text{max}})]$:

$$V \equiv \{(x, y) | x \in W_\Delta \land y \in W_\Delta, x_{\text{min}} \leq x \leq x_{\text{max}}, y_{\text{min}} \leq y \leq y_{\text{max}}\}$$  \hspace{1cm} (4.4)

so long as $\{(x_{\text{min}}, y_{\text{min}}), (x_{\text{max}}, y_{\text{max}})\} \in W_\Delta$. Used when referring to the visible bounds of the scaled *work*-space.

**user-space** The set of user accessible points on screen:

$$U \equiv V \cap M$$  \hspace{1cm} (4.5)

All user interaction with *view*-space (and thus *scale*-space) necessarily passes through this layer.

**Modifier** $ALT$, $CTRL$, and $SHIFT$ keys. Used in conjunction with numerous other keystrokes or mouse button presses to modify an action’s result or target.

### 4.2 General UI Overview

Most interaction with Visual Occam occurs through a high-level overview as seen in Figure 4.1. Here, the user is presented with a number of visual cues as to current program status along with detailed descriptions and tools through which source may be generated. Influences from a number of different development and typesetting utilities can be clearly seen, as per user familiarity and expectations analysis during the design phase (see section 3.2.4).

#### 4.2.1 Menu and Tool Bars

The menu, seen in Figure 4.2, contains an almost complete set of actions performable in Visual Occam divided into commonly accepted categories while the tool bar below holds only those considered to be the most commonly used. Categorical organization is as follows:
File  File operations including the creation, saving, importation, and closure of sources or projects. Application control is also merged into this category as per custom.

Edit  Standard code or content editing commands.

Source  Current source control. Includes commands to generate final occam code from graphical layout.

Library  Library and component catalog control. Allows user to select, create, or add content to specific libraries of procedures.

View  Commands for modification of current view or zoom level.

Help  occam  Reference manual and general application information.

It is worthy to mention that while there will always be a number of modifiers or gestures that cannot be easily translated into menu items, most major tasks have some presence. Furthermore, tool bar items not only share the look-and-feel of their menu
counterparts but a large number of these commands may also be accessed through the keyboard shortcuts indicated in the menu system. Of course, certain commands can only be accessed while the application is in the correct state: it does not make any sense to attempt a \texttt{CUT} command with nothing selected. As such, menu or tool bar items will deactivate and turn gray when not usable. Finally, standard shortcuts from other applications share equivalent functionality in Visual Occam where possible, though single-source commands that easily translate to the project level can be performed by appending the \texttt{SHIFT} modifier key to the appropriate shortcut.

4.2.2 Main Panel

The main panel (Figure 4.3) is central workspace of Visual Occam: a visual representation of your currently usable layout (\textit{view-space}). It is here that procedures and channels are defined, strung together, and from which \texttt{occam} code is generated. Originally envisioned as a single panel that switched content based on application needs, multiple design iterations eventually saw the creation of a tabbed system through which a user may modify multiple layouts at once, with the tabs themselves representing collections of components contained within a common procedure. Basic interaction involves the selection of modes through the \textit{tools palette} (see section 4.2.6) followed by either clicking or dragging across the panel. As prescribed under the usability design (section 3.2.4), drag-and-drop functionality also weighs heavily here to make the user’s experience more fluid: components may be dragged around, changing their displayable location, or even dropped in from outside sources (other tabs or the \textit{component library} (as described in section 4.2.5)). The main panel itself is, of course, also scrollable and zoomable, allowing the user to quickly move around the entire \textit{work-space}.
4.2.3 Minimap Panel

As per design considerations in section 5.2.2, the minimap panel (Figure 4.4) allows the user to more elegantly keep track of their viewport’s location (view-space) within work-space without hiding or obscuring the main view. Two layers are used: the bottom layer, containing a rendition of the entire work-space and the top most, which holds a representation of the view-space. Both layers are kept to scale, providing the user with far better visual orientation cues than the topologically sparse graphics of the main panel could ever hope to accomplish alone. Furthermore, the minimap itself is interactable, allowing the user to change view location or zoom by means of the same motions and commands as used on the main panel. This fluid control mechanism feels natural, allowing most users to quickly operate within even the most complicated of layouts. Further fine grained control is given by a right-click pop-up menu that allows easy access to many view-oriented commands.

4.2.4 Code Outliner

The code outliner, seen in Figure 4.5, represents a second style of overview designed to help the user keep track of the layered code generated by the application (again, as per the discussion in section 5.2.2). Not unlike the minimap panel, it dis-
plays the complete structure of the code and provides visual cues for current location or depth within the tree. Users are free to select any component from the outliner as if it was in the current scope, open such selected components for modification, or even change the structure by dragging and dropping components onto different levels. A good deal of effort was put into attempting to predict what kind of operations any given user would expect from this particular UI element, spurring a large number of design choices that would later be used to improve Visual Occam as a whole. This level of user interaction and the time put into ensuring that control of the interface feels fluid reflects deeply on a commitment to usability; there is no need to waste screen space on something that serves little purpose or is unusable.

4.2.5 Code Component Library

As the conceptual cornerstone of Visual Occam, the component library (Figure 4.6) allows a user to quickly reuse previously written code either from the current project or from a saved library. A user may select from the automatically populated list
and either drag the component to the desired location or simply invoke the INSERT command. The creation of such libraries is just as natural, with the interface readily accepting drag-and-dropped procedures (and thus whole networks) from the main panel. Libraries themselves, of course, can be saved to disk for later use, modified, merged, and even outright deleted.

4.2.6 Tools / Modes Palette

Shown in Figure 4.7, the tools palette allows the user to quickly change between interface states (referred to as modes during implementation) and perform the basic actions required to construct an occam layout. State-change buttons here function as toggles, giving the user the ability to quickly repeat similar creation or selection tasks without having to continually confirm the action. Still, while general control by means of clicking or dragging across the main panel is a feature of all modes, intricacies necessitate more detailed descriptions.
4.2.6.1 Select / Move / Resize

This mode, as it pertains to all three source types, is arguably the most complicated of the potential application states. In it, a user may elect to select, modify, or resize any given component by performing what should be a natural set of gestures, essentially operating as a contextually sensitive combination of the three actions. Of course, this kind of conglomerate requires more complete examination:

**Selection**  The default substate for this particular mode, selection occurs under the following circumstances:

- The user has nothing selected and clicks in a single location. The selection set is cleared and all components that contain the point are added to the set of selected items. Alternatively, a user with something already selected may click in a location that is not contained within the current selections to perform a similar selection.

- With nothing the user clicks anywhere within *work-space* (including other components) and drags to form a box. The selection set is cleared and all components that intersect with this selection box are added to the set of selected items. If something is already selected, the user may still chose to perform selection, but must necessarily start the drag process outside of any other components.

The *CTRL* modifier overrides this behavior and performs AND-selection through clicking and XOR-selection through the selection box. No distinction is made at this level between component types—procedures, nodes, or channels may all be selected at
once. Once selected, objects are highlighted in red and may be modified by means of the properties panel (see section 4.2.7) or further manipulated in this mode, depending on where and how a user next clicks.

**Movement**  Objects that are selected may be moved by simple clicking and dragging motions, though the user necessarily has to initiate the action from a point wholly contained within the target object (edges do not count here). As this is not necessarily clear in certain scenarios, the mouse cursor changes to the movement icon, seen in Figure 4.8.b, as a visual prompt (compared to the normal cursor in Figure 4.8.a). After initiating the click and during the dragging process, a dashed representation of the changes to be made are further shown to the user. Finally, upon releasing the mouse, the system attempts to move the object to the indicated position. Object specific implementations of this system are detailed as follows:

**Procedure**  Procedures are not allowed to overlap, releasing the mouse button will test to see if the new position is valid before committing the change. If the location is deemed unsuitable the entire operation is canceled. Channels that are connected to this procedure’s nodes are updated if a location change occurs, as are the nodes themselves.

**Node**  Node movement is necessarily confined to its parent procedure. Dragging around the work-space generates a preview that is confined to the edges of this procedure. As nodes are allowed to overlap, no check is performed after mouse release and the position is immediately committed. Furthermore, any channels connected to these nodes will have their paths automatically updated to the node's new position.

**Channel**  As channels are defined by path points instead of a whole shape, their movement preview is intrinsically different than that of nodes or procedures. Upon dragging, a preview of path is shown instead, constantly updated until mouse release. As with nodes, channel path point overlap is allowed and changes
are committed immediately upon release. As a further note, path point order is maintained during this process, allowing the user to create complex paths if desired.

Modification Finally, changes to components may also be initiated while in this mode, the exact details of which vary wildly by selected type and modifier depressed:

Procedure Assuming the user is operating on a selected procedure, clicking on an edge (indicated by 4.8.c–f) and dragging enables a resize operation. Resizing is constrained to the edges originally clicked on, the exact orientation (and axis of possible movement) indicated by the resize icons. A preview box, not unlike that used during procedure movement, is provided during the dragging process and releasing the button will commit changes in line with procedure movement.

Node Double clicking on a node toggles between input or output type. If this node is connected to a regular (non-shared) channel, the paired node also toggles. If this node is connected to a shared channel no further action is taken.

Channel Channels may be modified by either adding to or deleting points from their defining path. Addition is performed by holding down the CTRL modifier, clicking in any location, dragging, and finally releasing. As per other modes, a quick preview of where the new point will be placed is generated, the new path arrangement determined using a quick least-distance algorithm (see Appendix A, Algorithm 1.1). Path points, including those created by this addition, may be deleted by holding down the SHIFT modifier and clicking on or near the point when the mouse cursor changes to the icon seen in figure 4.8.g. Path order, in the deletion case, does not change.

---

2Defined as two successive clicks without much motion, the exact implementations of this depending on the virtual machine running the application.
Unlike the selection, movement, and resize tool, each of the creation tools operate independently to ensure that it is impossible to accidentally create a procedure in node creation mode or anything of the sort. Thankfully, though, all four have enough similarities that some meta-mode discussion is viable: all loop through operational stages (allowing quick repetition of tasks without requiring additional interface interaction), all allow user cancellation of the operation at any time through the escape button, and so on. Specifics for each creation mode are as follows:

**Procedure** A new procedure may be defined in one of two ways. Primarily, a user clicks and drags out a creation box that defines both location and extent of the procedure. Additionally, it is possible to simply click in the desired location to generate a minimum-sized procedure (specifically chosen to avoid situations where a user may inadvertently create a bounding shape of 0 width or height). As per selection, it is impossible to create a procedure that overlaps another. Attempts to do so will prompt the user with an error message and then revert back to the initial creation state.

**Node** Once in this mode a user may create nodes by clicking within a procedure, though important implications of Visual Occam’s tiered code structure (see section 5.3) come into play: nodes themselves are considered to be members of procedures and as such the user must be properly prompted to avoid confusion surrounding association or membership. To remedy potential confusion, a visual cue is given in
the form of an *object-of-interest* blue highlight. Generally speaking, this highlight will appear around the procedure best identified as “underneath” the mouse cursor and all nodes created will be assigned to this procedure. Since nodes themselves are restricted to the containing procedure’s edge, a preview of the node’s expected final position, represented by a dashed circle, is used to ensure that the user is comfortable with the result. Finally, although all nodes are created as output types by default, the user may choose to instead create an input type by simply holding down the *CTRL* button while clicking.

**Channel**  Basic channel creation starts when the user clicks on a viable node (again, *object-of-interest* highlighted when underneath the mouse cursor) to serve as the base point for the channel’s path. The user may then choose any number of intermediate points for the path by clicking anywhere on the main drawing surface, though holding down the *SHIFT* modifier to constrain new points to previous point’s (or node’s) *x* or *y* coordinates. A preview of the defined path is provided to help in creation, though nothing is committed until the user specifies a node (of opposite input/output type) to serve as the path’s end. As before, such nodes are highlighted in blue whenever the user moves the mouse cursor over a viable target.

*n : m* **Channels**  Any number of additional input or output nodes may also be attached to an existing channel. Creation of such shared channels follows the same steps as regular channels (*start point* → *path point(s)* → *end point*) with the exception that the final point must be a channel path point. It is important to note that these branches are parented to the original path, but are considered somewhat independent of the original channel and may be deleted or modified without disturbing the original. Viable nodes for connection are *object-of-interest* highlighted to ease this process.

**Group**  The grouping of objects (procedures, nodes and channels) is, of course, one of the major focal points of Visual Occam’s functionality. In order to facilitate user control over such collections three distinct methods of creation are provided:
Continuous If nothing is selected, the user may click and drag a creation box around any number of objects. Upon release of the button, Visual Occam will attempt to construct a valid group (the dimensions of which are determined by the creation box bounds) out of those objects selected during this operation. Objects that could not be added are left unmodified.

Individual Should the user be in grouping mode with objects already selected, Visual Occam runs under the assumption that the current selection set is to define the contents of a new group. It is possible to modify the selection set, even if empty, by holding the $CTRL$ button while either clicking on or dragging around objects. If no modifier key is being held, clicking and dragging will produce a creation box similar to that used in procedure generation. Upon releasing of the mouse button a new group will be created and the selected objects will be transferred.

Addition If the only object selected is a group of other objects, holding down the $SHIFT$ modifier while either clicking on or dragging around objects will add the newly selected object into the original group.

It is important to note that two channel restrictions are invoked to avoid potential hazards unique to Visual Occam’s tiered layout: channels must have at least one connected node in the group to be added and if all connected nodes of a channel are in the group so is the channel (this occurs automatically). This is done to prevent potential orphaning of channels in tiers where neither of their connected parents reside.

4.2.7 Property Panels

Aside from simply creating objects or manipulating objects, it is sometimes necessary to be able to directly modify the great number of properties associated with each object. These dynamic elements allow quick access to key controls, even those certain operations that have no clear visual analog, and are engineered to not overwhelm the user with an overload of information through clever use of resizing panels.
Furthermore, for those users with some level of experience with other integrated development environments, these panels maintain the designs seen time and again for such editing. Of course, due to the divergent nature of the various object types, three different property panels are necessarily developed:

**Procedure** Those properties directly related to procedures (both source and collection) and related members. Seen in Figure 4.9.

**General** Basic control information

- **Name** The name of the current procedure
- **Source File** The name of the current source file\(^3\)
- **Source** The current source associated with this procedure\(^3\)

**Advanced** Advanced properties not always necessary for network description.

- **Barrier** Name of a barrier for enrollment
- **Multiplex** Replication options that control creation of multiple instances of the same procedure

**Location** Visual and descriptive information.

- **Size** Width and Height of the procedure, in *work*-space coordinates
- **Position** *work*-space coordinates of the procedure’s upper-left corner

**Node** List of nodes and their properties.

- **ID** ID Number for the node
- **Name** Name of the node
- **Connection Type** Boolean, input or output node type
- **Display Type** Boolean, whether or not node name is displayed
- **Location** Location of node name or ID to be displayed
- **Data Type** Type that this node either accepts or transmits

\(^3\)Not visible for collection procedures
Figure 4.9: Visual Occam Source Based Procedure Properties Panel.

**Node** Those properties directly related to node as seen in Figure 4.10.

- **General** Basic control information
  - **Name** The name of the current node
  - **Type** The type of data expected by the node
  - **Input** Whether or not this node is an input or output node
  - **Display** Whether or not this node displays its name

- **Location** Visual and descriptive information.
  - **Location** Location of node name or ID to be displayed
  - **Position** work-space coordinates of the node’s center

**Channel** The connected channel of this node

**Channel** Those properties directly related to channels, Figure 4.11.

- **General** Basic control information.
  - **Name** The name of the current channel
Figure 4.10: Visual Occam Node Properties Panel.

**Type** The type of data transmitted by the channel

**Node** List of connected nodes and relevant properties.

**Procedure** Procedure of the node

**Name** Name of the node

**Input** Boolean, input or output type of the node

**Path** Visual and descriptive information.

**Number** Order this path point is operated on

**Path** List of work-space coordinates of the channel’s complete path

The properties panel itself is populated by any number of these sub types whenever objects are selected. Should nothing be selected, the properties of the main (controlling) procedure are displayed instead.

### 4.2.8 Output Tabs

Finally, the last major UI section, as seen in Figure 4.12, serves the small but important role of user prompting: providing a centralized location for system messages or general output. Most prompts and warnings will appear here, giving the user quick textual cues for their current actions or whatever error messages may pop up as a result of some action. Moreover, the tabs that denote the two major message types (System or Error) turn red whenever there is a new message on display, providing
a subtle visual cue that attention will eventually be needed in these tabs without necessarily distracting a user with constant popup messages.

4.3 Example Work Flow

Putting it all together, the following example shows, with both work-space visualizations and the resulting automatically generated occam code, just how quick and easy it is to produce networks of processes that would very quickly overwhelm a purely source driven environment. We begin by constructing the necessary base components for digital circuit emulation in occam, processes that correspond to logical
PROC gate.MUX (CHAN BOOL in?,
CHAN BOOL out.a!,
CHAN BOOL out.b!)

BOOL x:
SEQ
in ? x
PAR
out.a ! x
out.b ! x
:

(a) Code.

(b) Network.

Figure 4.13: occam implementation of a 1:2 MUX gate.

PROC gate.AND (CHAN BOOL in.a?,
CHAN BOOL in.b?,
CHAN BOOL out!)

BOOL x,y:
SEQ
PAR
in.a ? x
in.b ? y
out ! x AND y
:

(a) Code.

(b) Network.

Figure 4.14: occam implementation of an AND gate.

AND (Figure 4.14), OR (Figure 4.15), and NOT (Figure 4.16) operations, while noting that as occam channels consume data upon exchange such emulation necessarily also requires a data multiplexer (Figure 4.13) for each bit that must be transmitted to more than one gate.

With these base components, it becomes possible to construct more advanced processes, such as one emulating NAND logic (Figure 4.17). Furthermore, these NAND gates themselves may be strung together to create a network that operates as XOR (Figure 4.19).

Expanding this concept more, we put the previously defined gates together to form
PROC gate.OR (CHAN BOOL in.a?,
CHAN BOOL in.b?,
CHAN BOOL out!)

BOOL x,y:
SEQ
  PAR
    in.a ? x
    in.b ? y
    out ! x OR y

(a) Code.
(b) Network.

Figure 4.15: occam implementation of an OR gate.

PROC gate.NOT (CHAN BOOL in?,
CHAN BOOL out!)

BOOL x:
SEQ
  in ? x
  out ! NOT x

(a) Code.
(b) Network.

Figure 4.16: occam implementation of a NOT gate.

PROC gate.NAND (CHAN BOOL in.a?,
CHAN BOOL in.b?,
CHAN BOOL out!)

CHAN BOOL x:
PAR
  gate.AND(in.a?, in.b?, x!)
  gate.NOT(x?, out!)

(a) Code.
(b) Network.

Figure 4.17: occam implementation of a NAND gate.
PROC gate.XOR (CHAN BOOL in.a?, CHAN BOOL in.b?, CHAN BOOL out!)
CHAN BOOL a,b,c,d,e,f,g,h,i:
PAR
  gate.MUX(in.a?,a!,b!)
  gate.MUX(in.b?,c!,d!)
  gate.NAND(b?,c?,e!)
  gate.MUX(e?,f!,g!)
  gate.NAND(a?,f?,h!)
  gate.NAND(b?,g?,i!)
  gate.NAND(h?,i?,out!):

Figure 4.18: occam implementation of a XOR gate.

![Diagram of XOR gate]

Figure 4.19: occam network of a XOR gate.
a single bit adder (Figure 4.21), four single adders into a four bit adder (Figure 4.23),
and then two four bit adders to create a full integer adding circuit (Figure 4.25). As
should be obvious, the total amount of time involved in creating this adder spanned
but a few minutes, even though it contains over 600 parallel processes (as per Table
4.1). Furthermore, it is easy to check for correctness of the system by simply visually
following the paths the channels take. As the generated occam code shows, even in
the trivial case, such checks quickly spiral out of control as operations and parameters
quickly build up. Without tools to properly visualize and construct these networks,
anything much larger would become too much of a burden to deal with.

<table>
<thead>
<tr>
<th>Network</th>
<th>Figure</th>
<th>Channels</th>
<th>Processes$^4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUX</td>
<td>4.13</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>AND</td>
<td>4.14</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>OR</td>
<td>4.15</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>NOT</td>
<td>4.16</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>NAND</td>
<td>4.17</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>XOR</td>
<td>4.19</td>
<td>13</td>
<td>29</td>
</tr>
<tr>
<td>ADDER</td>
<td>4.21</td>
<td>37</td>
<td>79</td>
</tr>
<tr>
<td>ADDER.FOUR</td>
<td>4.23</td>
<td>151</td>
<td>316</td>
</tr>
<tr>
<td>ADDER.FULL</td>
<td>4.25</td>
<td>303</td>
<td>632</td>
</tr>
</tbody>
</table>

Table 4.1: Constructed network cumulative complexity analysis.

$^4$Excluding constructing process such as SEQ or PAR.
PROC adder.SINGLE (CHAN BOOL in.a?, CHAN BOOL in.b?, CHAN BOOL in.c?,
CHAN BOOL out.s!, CHAN BOOL out.c!)
CHAN BOOL a,b,c,d,e,f,g,h,i,j,k:
PAR
  gate.MUX(in.a?,a!,b!)
  gate.MUX(in.b?,c!,d!)
  gate.XOR(a?,c?,e!)
  gate.MUX(e?,f!,g!)
  gate.MUX(in.c?,h!,i!)
  gate.XOR(f?,h?,out.s!)
  gate.AND(g?,i?,j!)
  gate.AND(b?,d?,k!)
  gate.OR(j?,k?,out.c!)
:

Figure 4.20: occam implementation of a single bit adder with carry.

Figure 4.21: occam network of a single bit adder with carry.
PROC adder.FOUR (CHAN BOOL in.a0?, CHAN BOOL in.a1?, CHAN BOOL in.a2?,
CHAN BOOL in.a3?, CHAN BOOL in.b0?, CHAN BOOL in.b1?,
CHAN BOOL in.b2?, CHAN BOOL in.b3?, CHAN BOOL in.c?,
CHAN BOOL out.s0!, CHAN BOOL out.s1!,
CHAN BOOL out.s2!, CHAN BOOL out.s3!,
CHAN BOOL out.c!)
CHAN BOOL a,b,c:
PAR
  adder.SINGLE(in.a0?, in.b0?, in.c?, out.s0!, a!)
  adder.SINGLE(in.a1?, in.b1?, a?, out.s1!, b!)
  adder.SINGLE(in.a2?, in.b2?, b?, out.s2!, c!)
  adder.SINGLE(in.a3?, in.b3?, c?, out.s3!, out.c!)
:

Figure 4.22: occam implementation of a four bit adder with carry.

Figure 4.23: occam network of a four bit adder with carry.
PROC adder.FULL (CHAN BOOL in.a0?, CHAN BOOL in.a1?, CHAN BOOL in.a2?,
CHAN BOOL in.a3?, CHAN BOOL in.a4?, CHAN BOOL in.a5?,
CHAN BOOL in.a6?, CHAN BOOL in.a7?, CHAN BOOL in.b0?,
CHAN BOOL in.b1?, CHAN BOOL in.b2?, CHAN BOOL in.b3?,
CHAN BOOL in.b4?, CHAN BOOL in.b5?, CHAN BOOL in.b6?,
CHAN BOOL in.b7?, CHAN BOOL in.c?, CHAN BOOL out.s0!,
CHAN BOOL out.s1!, CHAN BOOL out.s2!,
CHAN BOOL out.s3!, CHAN BOOL out.s4!,
CHAN BOOL out.s5!, CHAN BOOL out.s6!,
CHAN BOOL out.s7!, CHAN BOOL out.c!)

CHAN BOOL a:
PAR
adder.FOUR(in.a0?, in.a1?, in.a2?, in.a3?, in.b0?, in.b1?, in.b2?,
in.b3?, in.c?, out.s0!, out.s1!, out.s2!, out.s3!, a!)
adder.FOUR(in.a4?, in.a5?, in.a6?, in.a7?, in.b4?, in.b5?, in.b6?,
in.b7?, a?, out.s4!, out.s5!, out.s6!, out.s7!, out.c!)
:

Figure 4.24: occam implementation of a full adder with carry.

Figure 4.25: occam network of a full adder with carry.
CHAPTER 5

EXPERIENCES, THOUGHTS, PROBLEMS AND SOLUTIONS

Given the length and depth of development required for the Visual Occam system, it should not be a surprise that a nontrivial number of issues and considerations were encountered. While many of these gave enough pause to warrant some discussion, there is little need to explore the nuances of each conundrum. Instead, a select few of the most influential problems, ranging from basic design flow patterns to language and API concerns, are presented as an acknowledgment to the kind of thought processes required to overcome development challenges.

5.1 Design Iterations

The process by which we develop software is, fortunately or not, still a burgeoning field. Methodologies that themselves range in scope from the smallest projects to the largest of scales are constantly evolving as both the industry and science behind it mature. As such, it would be presumptuous to say that any given approach for development is ideal without necessarily delving into a rigorous analysis of the project at hand. That being said, such analysis could easily sum to be the body of an entire discourse and simple, albeit subjectively chosen, techniques would most likely cover all bases sufficiently. Thus, it is from the author’s experience that the various iterative techniques (be it the Rational Unified Process, Agile Development, or even SCRUM) are proven enough to provide a workable foundation for the project. Of course, detailing the full set of iterations and phases used for Visual Occam would be foolhardy, at best; instead, a coarse-grained example of the development process for a single interface element is used. Without loss of generality, these techniques helped guide the evolution of not only the interface but the code-base as a whole.

5.1.1 Conceptualization

Before any code work may be done, a series of rough concepts are drawn out. These hand drawn images (like that in Figure 5.1) are easy to produce, allowing a
large number of ideas to be hammered out in the shortest amount of time possible. General utility and layout of the interface element are explored to great lengths with these graphics while finer grained concepts (what individual components do, names, or even labels) are set aside for later iterations and phases.

5.1.2 Rapid Prototype

From the conceptualization phase, the best candidates are chosen and quickly assembled into a semi-functional prototype (Figure 5.2). Names, labels, attributes, and other details omitted previously are added - though not necessarily populated with useful information. Rather than waste time fully developing interface elements that may potentially be discarded, this phase of work seeks to answer questions of aesthetics and feasibility in the unfortunate event that the ideas might simply be too cumbersome to use or unwieldy to translate into a working codebase.

5.1.3 Iterative Development

After a number of prototypes and potential returns to high-concept creation, the best ideas are put together into a functional piece of code and integrated directly with the rest of the system (Figure 5.3). Here, interface elements are given utility
Figure 5.2: Object properties panel design prototype.

and general testing begins to iron out usability issues not originally caught in previous phases. Major defects, if found, may spur the creation of new solutions through more prototyping, but, barring the unforeseen, tend not to affect general layout or expected function.

5.2 Interface Usability

During Visual Occam’s conceptual design, three major usability concerns tended to come up time and again.

5.2.1 Anticipating Actions

Having roots in other, far more mature, graphical integrated development environments, Visual Occam suffered from an almost project paralyzing inability to properly predict what actions, if any, a user would most likely want to perform. With each design iteration, the total possible number of user operations fluctuated rapidly making it further difficult to pin down the exact flow of actions possible (or those that should be possible). A novice user would be certainly comfortable operating the interface by whatever specified course of action was originally laid out, but the experienced
user would invariably bring preconceived notions of what mouse motion or keystroke should do what.

Naturally, while it would have been completely impractical to attempt to appease all user wants, allowing a user to go through the same motions in Visual Occam as one would in Netbeans to perform similar tasks is necessarily a great boon towards familiarity (and thus usability in general). Thus, a compromise was made: Visual Occam would attempt to invoke as many potential actions and avenues for access to those actions as would be reasonable under current time frames. Each prototype interface underwent at least some modicum of usage testing, those motions that seemed unnecessary (or, of course those that were lacking) noted down and potentially later implemented. What results from this, at least before more rigorous testing, is an interface that should be familiar enough to most users that general work flow patterns may be established - different for each person if necessary.

5.2.2 User Orientation

Generally speaking, the visualization of code in the work-space is topologically sparse: most views contain surprisingly few visual features by which a user may orient themselves. As such, it becomes unfortunately easy for a novice user to get lost amidst
the myriad of processes, channel paths, and nodes. With proper implementation overviews and zoomability, however, it is certainly possible to minimize these effects.

**Zoomability** Zoomable User Interfaces (ZUIs) have been picking up steam not only in the world of mobile computing, where screen real estate is a premium, but also in desktop computing as an alternative to multi window setups. By allowing the user to quickly transition from various *scale*-spaces using natural motions, spatial orientation issues and the errors that come with the territory are minimized.

**Overview** Despite the sparse nature of the layout, large (and potentially complex) networks are certainly possible to construct. With a large or deep enough network and small enough screen resolution, the user-space may become a bottleneck in usability. To combat this two different types of overviews are provided: geometric and semantic.

**Geometric** Described as an overview or minimap in the literature, this view provides a to-scale representation of both the entire *work*-space and its relation to *view*-space. Use of this overview allows a user to quickly reorient themselves after zooming or even traverse large network spaces.

**Semantic** Depth issues are directly handled by giving the user a complete structural map of the network hierarchy in the form of a tree-view, or outliner. The user is able to quickly switch between views of different subnetworks without necessarily having to identify each individual parent along the way.

In tandem with the zoomable user interface, these two views of the given spaces allow for visualization of even the most difficult object or data flows and provide for a much more usable environment.

**Multiple Window View** Though certainly not mutually exclusive, multiple view interfaces are most easily compared with zoomable user interfaces in terms of user benefit. In general, it is relatively obvious that the more comparisons must
be made of different views, the more benefit adding these addition view interfaces becomes. That being said, however, it is rare to directly compare networks in Visual Occam. Instead, internal network operations are treated as a black box (indeed, this is the entire essence of CSP like network behavior anyway). Furthermore, given the general lack of interconnectivity between system layers due to hierarchal structure after effects, there is little to be gained from having multiple views of the same spare topology. Still, it cannot be denied that the ability to quickly jump between views is useful. As such, a compromise is made by means of tabbed panels. Visual Occam allows the user to open as many (as fit in memory, naturally) tabs of as many different networks as is necessary, even if only of the same objects, but provides no mechanisms to place both views side by side. If a user wants to compare connections made or the like, it is necessary to switch tabs. Since there is not much to an individual network, the amount of memorization required by the user in this task is deemed acceptable.

5.3 Program Structure

It was decided during the initial planning of Visual Occam that the best way to handle the translation of graphical representations into occam code would be to treat the entire system in much the same manner other, more traditional, compilers do: process across syntax trees. We thus define the following tree elements as Visual Occam objects:

```
     C
    / \  
   B   D  
   / \    
  A   E   
```

Figure 5.4: Object hierarchy visualized as a tree.

**Figure 5.4.(A) Node** A process. Visual Occam views occur from this perspective.
Objects A, B, and C are not visible to each other.

(a) Objects A and B may be directly connected.

(b) Objects A and B may be directly connected.

Figure 5.5: Object hierarchy visibilities issues.

Figure 5.4.(B) **Edge** Relationship by aggregation; parent nodes (where an edge originates) necessarily contain the child nodes (indicated by edge termination).

**Figure 5.4.(C) Root** The master process in which an entire occam project is contained. A direct analog to Java/C’s **MAIN** procedure.

**Figure 5.4.(D) Inner Node** A process network, specifically referred to as a collection in the Visual Occam literature.

**Figure 5.4.(E) Leaf** A process defined strictly with source code.

Of course, as the occam language is itself not constructed in a purely hierarchal fashion (it should be evident from the formal description (see section 2.3.1) that all specifications may occur at the same level), this kind of definition naturally adds subtle translation nuances to the system. For one, while it possible for any node to be connected by means of a communications channel, not all nodes are visible in each view; in terms of our defined tree structure, nodes must necessarily share not only the same level but the same parent to be considered visible to each other (Figure 5.5).

We combat this issue by realizing that an unconnected node within a process network may be best treated as a representation of external influence. In simple terms, unfilled communication requests in a child node are also unfilled for the whole
network; in each such situation it is simple enough to generate a corresponding for-
mal parameter in the parent’s specification to pass connection information through
the network structure. By definition, such parameters will naturally propagate up
the entire hierarchy until the request itself is fulfilled with a communication channel
thereby connecting the originating node. Thankfully, this kind of behavior is some-
what in line with CSP’s (and thus occam’s) goal to produce loosely coupled process
networks. By strictly adhering to a defined hierarchy it is impossible for any network
to be connected to another outside of those nodes where either both are visible or at
a common ancestor.

Of course, as the whole point of this structural organization is to provide an easy
(or, at least, correct) mechanism for the generation of occam code for corresponding
visual networks, we note that it is clearly necessary to visit each tree node in order to
fully describe the behavior a connected system. Furthermore, we note that as all the
process specifications are placed at the same level regardless of their hierarchal order
in the tree but should come before the process that instantiates them, the choice of
tree traversal pattern falls most simply to a preorder depth first traversal. Taking
a note from similar systems in Java, it is thus clear that the application of a visitor
pattern (even if a heavily modified one) is sufficient for these purposes.

5.4 Visual Occam Limitations

Visual Occam has a number of limitations when it comes to the generation of
occam networks that are not necessarily clear from the onset. Although partially
covered in the discourse on hierarchal structure, many other limitations come as
either a direct consequence of implementation or result from choices made during
design that later reflected in implementation.

While at a cursory glance it would be simple to call each node independent of
each other node, complications arose when considering what level of checking to do
with communication channel connections. Strictly speaking, it is outside the scope
of Visual Occam to perform a full syntax check on generated occam code, but some
‘easier’ checks might prove to help the user produce cleaner and compilable code more efficiently. As such, we note what kind of restrictions apply to the connection of two nodes for communication:

**Connection Type** Single connection channels (one-to-one communication) can only be defined between an output node and an input node. Changing the connection type of any node connected in this fashion must also change the type of the other, paired node.

**Data Type** Nodes must be of the same type when connected. Changing the data type of any node connected in this fashion must also change the type of the other, paired node.

Both of these two simple tasks, if adhered to in the Visual Occam interface, are used instead of rigorous type checking and guarantee that the code naturally generated by Visual Occam will at least be error free in this respect. Type checking in the user input process code, however, is not performed and may (invariably) produce errors but is not the focus of the program at large.

Like nodes, channels themselves add a bit of non-trivial complexity to the composition of the Visual Occam structure. Though it is should be clear that the connected nodes and procedures referenced by channels should not have ownership of the aforementioned channel, channels themselves need to be placed somewhere. Furthermore, there is existential confusion regarding channels themselves: a channel with no nodes (both input and output) is functionally identical to no channel at all. It is therefore assumed that the existence of any channel is directly related to the existence of the connected nodes

**Existence** Functionally speaking, there is no difference between a channel with no nodes (either input or output) and the outright absence of of a channel. There is, therefore, a syntactic entanglement between channels and nodes (and thus procedures); to rectify this ambiguity, no channel may be created without first
defining connection nodes (order itself here should not matter). This further ensures that at any given moment Visual Occam is in a state where it can generate valid code for the channel.

**Encapsulation** As with all of the Visual Occam constructs, it is necessary to attribute ownership of channels to another object. In natural *occam* code, channels may be written as parameters to a series of procedure calls; Visual Occam attempts to extend this behavior into the object space by giving ownership to the single procedure that contains (or calls) the connected procedures.

**Shared Channels** n-m channels prove to further complicate issues that tend to arise with regular channels. While seen as extensions of the original channel definition, it is clear that a special set of rules need to apply (both for creation and general handling). Visual Occam specifically defines such shared objects over existing channels, requiring a valid input and output node to be first chosen. Upon connection of further nodes (either input or output), the regular channel is converted to the broader-scope shared channel. While this does add an extra step to the creation process, it reinforces the similarities between channels and shared channels in a way that a disjoint creation process could not. As a result of this, of course, it should be noted that no CLAIM checking is done on shared channel ends, it is up to the user to properly utilize those channels that are specified in the interface.

## 5.5 Swing

Though the Visual Occam project focuses almost exclusively on the broad strokes of general user interface development and the integration of *occam* into a new paradigm, proper coding simply could not have occurred without first spending a deal of effort in learning how to use those tools readily available for such design. Swing, then, as the de facto API for Java based graphical user interfaces, was scrutinized to great extent. There is no doubt that Swing is an incredibly powerful utility, but in the
attempt to create not only a useful but modern interface a number of issues came to light when faced with Visual Occam’s needs.

5.5.1 Versatility vs Power vs Expectations

Given the scope of the Swing/AWT API, it is possible to create just about any kind of interface component imaginable [MA05]. Unfortunately, as a result of this need to be versatile, the default component selection is itself meager at best. Everything that exhibits a modicum of specialization requires extension and modification—tasks that while not incredibly complex certainly add to the overall length of time necessary to construct interfaces. The most glaring example of such work came as a result of the desire to support multiple tabbed views in the Visual Occam environment. Such tabbed panes, being defined as a standard feature in the Swing API were easy to add, their management (ordering, opening, closing, etc), however, proved to require multiple classes, overrides, listeners, and events just to achieve par with what may be considered expected behavior. Unfortunately, it seems likely that Swing’s philosophy of overarching versatility and incredible power undermines the ability to easily manage extensions into modern design elements. Furthermore, as user expectations drift, the current set of default elements will quickly become outright archaic or simply too obtuse to use.

5.5.2 Layout Managers

Layout managers are the amazingly useful but two-sided sword of the Swing interface development platform. In almost all respects, the use of such managers allows a developer to quickly define layouts in a natural manner, without necessarily specifying absolute distances or positions. Furthermore, many of these layout managers automatically arrange their contents based on the size of their container, resulting in very versatile and flexible layouts. The catch, unfortunately, is that this paradigm is completely all-or-nothing; it makes no sense (nor is it possible) to attach multiple managers to the same object, even if only to control a subset of the layout. Naturally, it is possible to get around this kind of limitation by introducing nested containers, but this is not necessarily optimal for any given situation. Moreover, it can be dif-
difficult or downright impossible to find a layout manager that behaves in the exact way one envisions for some design. Compromises are quickly made and the chosen layout manager ends up defining the layout design instead of aiding in the process of development.

5.5.3 Non-Static Elements

Animation of elements, even if only switching between two runtime defined states, is a nontrivial task in the Swing environment and unfortunately highly coupled with the layout manager of choice. Though such animation is certainly not necessary, the subjective difference a fluid interface makes is most certainly an important matter: it is the creative use of such animation that allows collapsible elements and information to automatically readjust their neighborhood and prevent distracting lapses in interface flow (See Figure 5.6). Examples of such behavior may be seen in virtually all modern interfaces, to the extent of which it is surprising to see Swing not natively support such design.

5.5.4 SwingX

Given the desire to introduce more complicated, animated, or simply resizeable elements into the Visual Occam interface, it became necessary add SwingX into the development codebase as a general supplement to Swing proper. While a large amount of what SwingX provides is far outside the scope of projected needs, the potential amount of time saved by using the available technology was substantial; after only a short struggle with the integrated development environment to recognize the library, a much larger toolset, including not only collapsible panels but auto-completion mech-
anisms and increased support for table and tree components, becomes accessible. These predeveloped, sleek features allowed development of Visual Occam to proceed quicker than ever while maintaining more of the original design focus than with Swing alone.

Of course, it should be noted that that addition of such libraries of component extensions does add appreciable bloat to the system as a whole and it certainly requires time to get accustomed to the new API commands. Furthermore, the use of nonstandard components necessarily exposes general display and usability issues with Netbeans that would not have ordinarily been noticed. These issues are minor though in comparison to the utility gained and put aside as simple project growing pains.

5.6 Auto Generated Code

The code automatically generated by Netbeans for the graphical user interface was one of the major problems encountered when designing and writing Visual Occam. Despite the original design intentions of keeping code as loosely coupled as possible it is unfortunately easy to fall into the trap of simply using the Netbeans interface to modify components within the same file. Typically, integrated development environments such Netbeans generate code into very few (if not only one) source files. While it is trivial to keep the code well organized using tool specific collapsible / hideable comment blocks, it is certainly not considered good design. Furthermore, as the interface becomes more complex it becomes more likely that small changes will have dramatic effect on the generated code. As such, it would be foolhardy to directly modify much within the source created by these tools and a layered approach becomes necessary (see section 3.1.1). As indicated in the design document, there is a great concern that allowing too much coupling invariably leads to poor coding and other related pitfalls. To combat this, the interface layer was instead used to great effect.
CHAPTER 6

RELATED WORK

Given that development of Visual Occam touches on a rather large number of topics, a quick overview of work relating these many facets is provided.

6.1 Visual Programming

Research, and other such endeavors, regarding programming by manipulation of graphics instead of simply text [KMF89] spans now more than twenty years of work [Mye90, CKLI94, RKS98], delving into almost every facet of development. From everyday languages to embedded systems [JS08, KJ08, SJ08] to even VLSI design [SE87], development through visualization has proven an extremely effective paradigm (at least in highly specialized situations) time and again.

6.1.1 CSP

As the basis of occam, and thus the Visual Occam, CSP’s loosely coupled process networks have long drawn the attention of visual programming research. Though there are examples of complex visualizations as early as 1987 [DS87], the tools were only developed to interpret code for subsequent animations of the execution trace. Indeed, for the most part, CSP networks were still constructed by hand with minimal aide from what can only be considered primitive diagrammatic tools. It is not until Hildernick’s proposals [Hil02, Hil03] that we see that formal creation of graphical notation standards and eventually automated tools to ease the management thereof [BJ04]. These tools, while only basic proof-of-concept editors and admittedly not detailed enough for full fledged programming, clearly showed the power a graphical editor may have. Still, the results were not without flaws:

- Models in gCSP are, unfortunately, static. It is impossible to flatten complex trees of processes, bring individual elements into a different scope, or in general morph topology.
• Grouping notations with large hierarchies, at least as described by Broenink et al.[BJ04], proved cumbersome at best as the choice between visual clutter or ambiguity became more difficult.

• The user developed models are themselves not easily reusable. Though the mechanism exists it is hardly user friendly.

6.1.2 occam

Not to be overshadowed, two major works, namely VISO [AMA97, AMA98, AM00] and the Visputer [ZM94], have been also developed as steps towards bringing the visual paradigm into the world of occam. Both of these projects, however, focus almost exclusively on visualizing the entire occam language - from IF to PAR to ASSIGNMENT constructions. Such programming in-the-small, as noted by Gorlick [GQ94], while certainly a fruitful topic of research, ultimately misses the point when it comes to real world development:

• Systems, if to be deemed useful, need to be able to easily construct networks of hundreds, if not thousands of processes. While both occam utilities allow the creation of such large networks, the tools themselves do not provide the functionality to deal with such creations on a reasonable time table.

• Compositional errors should not be treated as mistakes but rather as requests for information or components. The interface should not punish the user, but rather attempt to compensate and reinterpret.

• Incomplete visualizations need not necessarily be discarded but, instead, treated as partial specifications for use in other systems.

6.1.3 Circuit Design

Unlike the visual programming methods used in software development, digital circuit design applications (specifically, electronic design automation tools) are far more mature [SE87]. From major commercial software ventures by Synopsys, Cadence, or
Mentor Graphics to open source projects such as gEDA or Magic, the entire computer design industry is thoroughly entrenched in visual modeling. Granted, while it is clear that space, heat, materials cost and other physical requirements have created a greater need for such tools, many facets of these utilities share direct analogs to software visualizations. Clear design choices necessarily follow:

- Without a doubt, given the size of the industry and the number of individual users of these utilities, usability is a primary concern. Operations are concise, instructions clear, and though a user may not necessarily want to construct a faulty model, they are certainly given the ability to do so.

- Electronic Design Automation tools are inherently divided in focus. Those that perform VLSI design do not worry about the inner workings of each integrated circuit and instead completely blackbox operation for definition in another, more specialized, environment. While this means that it is much more difficult to produce a complete circuit using a single tool, it also implies that appropriate visualization and handling is applied for each level of focus.

- Designs for digital circuits are meant to be reusable. Many (if not all) of these tools allow users to import existing models from a wide library selection (often even one tailored to the given project) and further allow the addition of user created models into other user created models.

6.2 Graphical Programming Environments

Modern integrated development environments, such as Netbeans [BGS02, Böc09], Eclipse [DFK+04], or Visual Studio [SF03], serve as a benchmark for real world visual development. Almost all of these tools, despite their somewhat broad scope, have mechanisms for the designing of User Interface elements through graphical environments. Furthermore, it is evident that design similarities run pretty deep in through these platforms:
Tabbed View Tabs, literally multiple views of code, are used universally in these environments as substitutes for multiple windows, even if it may be necessary to sometimes compare elements in one tab to that of another.

Outliner / Navigator An outliner or hierarchal display is provided to help a user quickly navigate between code structures and other nested elements.

Pallet Visual editing provides a pallet of tools and objects with which a user may “paint” the most commonly used interface elements.

Properties Properties of objects are collected into one location, providing quick and universal access to every mutable aspect of any given (typically by means of selection) object.

Output User output tabs are used to give verbose user feedback should the need arise.

6.3 Design and Usability

Aside from project specifics, interface design and usability concerns are staples in the field. So vast is the body of knowledge here [Tid05, SPCJ09] that it would, unfortunately, impractical to list all the related works. What can be said, however, is that even though the general concept of graphical user interfaces tend to be preferred and reduce the amount of training time required [ZAP94], the unfortunate wholesale ramifications of bad design [Jan98] inherently means that developers need to input a great deal of effort into design to ensure proper usability. Other development platforms, such as Coral [SM88], cite general power and ease of use as direct artifacts of the program’s structural hierarchy (to allow new mechanisms to be quickly added if necessary), judicious use of inheritance (to increase efficiency and reusability of code), and a general focus on user interface design. Even then, proposals for new paradigms in design are popping up all the time, ranging in scope from top down approaches to minimizing development upset time [SR95] to arguments for wholesale reconsiderations of structural interface elements [Mye05].

83
Moreover, as noted by Shneiderman [Shn00, Shn03], design towards usability must necessarily also take into account the variability of user: each individual will bring to the application a different set of environmental factors, work experiences, and general predispositions. What is considered usable for one such set is not necessarily usable by another and to be truly considered universally easy to use requires surmounting even more challenges than other literature tends to mention.

6.4 Visualization Paradigms

During the design phase of Visual Occam, proper handling of the work-space became a point of great concern as it was intended to give the user the ability to create massive networks (both in size and depth) without necessarily wanting to overload them with information. Thankfully, much like visual programming, data visualization has seen a veritable explosion of paradigms [CKB08, Shn08], including everything from focus manipulation to semantic zooming to simple multiple views. Recommendations, however, seem to point that a combination of zooming and overview support to provide the user with the most flexibility and performance gains while, at least for well designed systems, reducing overall mental load.

6.4.1 Overviews

Directly comparing zoomable user interfaces with and without overviews in a number of different scenarios, independent empirical studies involving desktop situations Hornbæk [HBP02] and smaller resolution mobile devices Büring [BGR06] show that although there is a bit of a trade off for satisfaction vs speed between such systems (at least on larger desktop environments, users tend to be faster without an overview but enjoy the experience less), the intuitive nature of the overview+zoom setup greatly benefits situations where it is difficult for a user to quickly orient themselves for otherwise. Furthermore, comparisons of different types of overview support for zoomable interfaces by Burigat [BCP08] confirm that users not only performed less reorientation correction, but were able to almost exclusively focus their work-space interaction (panning or zooming) on solving the problems at hand.
6.4.2 Multiple Window Interfaces

Similar comparative studies have also been performed regarding zoom capable interfaces that further support multiple views [PW06]. Taking an experimental approach to the problem, Plumlee and Ware created scenarios for users to solve using a number of different interface configurations. Although their experimental setup exposed a number of unknown variables that made definite conclusions difficult, it is clear that the cost of zoomable navigation and multiple visual windows depends greatly on environment and experience. In circumstances where users are required to compare large numbers / complex sets of objects (beyond, for example, their capacity for working visual memory) multiple views become an attractive feature [LHG04] while, on the other hand, those situations where users need only to occasionally commit to another window are often best treated with zooming.

6.4.3 Dual Visualizations

Further, there is growing support for the usage of multiple visualizations of the same data set to more properly describe networks [NSGS07], whereby combining multiple coordinated views, user productivity and information discovery seems greatly enhanced.

6.4.4 Visual Occam

Again, all of this is put together in the end to help better develop the Visual Occam system. By custom tailoring what views are used and how the user is allowed to manipulate everything, general usability concerns may be further appeased. In particular, we attach the following paradigms to the various user interface elements previous described:

**Overview** Through the minimap component, Visual Occam supports quick and easy reorientation in line with other overview supporting systems.

**Zooming** Giving the user control over scale, as typically paired with overviews, allows the user to spend less time worrying about the position of the view port and more actually performing useful actions.
Multiple Window Interfaces  Multiple windows are supported by means of tabs only, there is little need to compare networks and thus no need to further complicated the interface.

Dual View  Multiple visualizations, in terms of both the outliner and the main panel view, are provided to ensure that a user gets as much information in the smallest amount of space possible.
CHAPTER 7

CONCLUSION

In this discourse, we have shown the process by which a complete development platform for the visualization of occam networks has been developed, starting from basic design concepts to detailed implementation notes, and that through this tool it becomes clear just how powerful visual programming may be for networks of loosely bound processes. By using Visual Occam (or other such tools) even a novice user may benefit from the deep mathematical foundation CSP brings to parallel programming all while constructing vast networks of processes by means of only a few clicks or drags.

This aside, the project’s length development cycle allows us to get a far better understanding of what exactly it takes to produce a modern graphical user interface for such tasks:

- Design is paramount for well rounded interfaces and the general success of large scale projects. In nearly all problem situations, well thought philosophies and design patterns alleviated potential pitfalls while allowing the developer to maintain focus on the larger picture. Without the laboriously produced design documents and exhaustive iterative process through which Visual Occam grew, it is clear that the amount of work necessary to stay afloat would have been outright overwhelming.

- Event driven programming in an object oriented language using automatically generated tools almost necessarily precludes good coding practice. It is only through the careful design around the nature of such tools that coupling and cohesive standards were maintained.

- Swing tools are powerful, easy to learn, but difficult to master. Innumerable hours were spent attempting to mold the Swing tool set into the proper shape without breaking functionality.
Of course, the monumental effort it took to get Visual Occam finally operational not-withstanding, the end result of development is an application that seems to be quite capable of bringing the concepts of process oriented design together with the pragmatic approach of visual programming. Running contrary to normal expectations, it is almost trivial to handle networks of hundreds, if not thousands, of processes and channel interconnections by applying work flows specifically designed to be both agile and easy to understand. Indeed, the very prospect of furthering this kind of network design is very exciting and will most assuredly shape the way we handle large scale parallel programming in the future.
FUTURE WORK

Despite the amount of effort put into the design and development of Visual Occam, there exists a large volume of work to be done. A lengthy, but non-exhaustive, list of a few of the more interesting possible additions showcases just how much more potential development is needed in this seemingly never ending project.

**Source Input** Inputting the code for a source procedure completely within the Visual Occam interface is awkward for large enough definitions. Although the user is allowed to specify an external source file from which the program will pull code, it would be time saving and certainly beneficial to have a full-fledged editor at the user’s disposal.

**Expanded occam Support** Visual Occam does not currently generate the full spectrum of occam source. Aliases, mobile objects, and other more advanced features need to be added in to allow users to generate a complete gamut of networks.

**Revamped Control** The number of action / control options in the developed system is, while not insignificant or trivial, certainly incomplete. Refinement of the drag/drop system for network generation alone (with, perhaps, the ability to grab information directly from external sources) would most assuredly give Visual Occam a professional feel.

**Enhanced Options** A number of additional options, mostly to allow a user to customize the look and feel of the system would certainly help appease the potentially large number of users (with equally varying experiences using such editors). In particular, though, the following are deemed most interesting:

**Procedure Shapes** It is common in many diagrammatic environments to use different shapes to aide in quick differentiation. By allowing a user to
specify exactly how they want a procedure to be output (say, the difference between \textsc{and} and \textsc{or} gates) Visual Occam would provide a more fluid experience simply by giving better visual cues.

\textbf{Display Options} Display options for customizing various interface features, including thickness of lines, borders, and other subjective aesthetic choices could prove useful for further differentiation.

\textbf{Libraries} If Visual Occam is to be a true development platform, a large number of predefined process networks need to be created and placed within relatively easy to access libraries if only to promote the concepts of process oriented design.

\textbf{Usability Testing} Usability itself, while a great concern for the development of the Visual Occam platform, was never given the opportunity to undergo more rigorous use-testing. By proceeding through a more formal analysis of the choices made during design and development, it would be possible to generate a clear picture of what works, what does not, and which of the choices made had the most profound impact on usability as a whole.
Algorithm 1.1 Live Channel Construction

Require: $\rho \equiv (\rho_x, \rho_y) \in W$
Ensure: $\rho \in path$

if $path \equiv \emptyset$ then
    $path \leftarrow \rho$
else
    $\beta \leftarrow \text{closest}(\rho, path)$
    $i \leftarrow path.\text{index}(\beta)$
    $n \leftarrow path.\text{size}() - 1$
    if $\beta \equiv path[0]$ then
        $path \leftarrow path.\text{insert}(0, \rho)$
    else if $\beta \equiv path[n]$ then
        $path \leftarrow path.\text{insert}(n, \rho)$
    else
        $\{ \beta \text{ not an endpoint} \}$
        $\alpha \leftarrow path[i - 1]$
        $\delta \leftarrow path[i + 1]$
        $A \leftarrow \text{dist}(\alpha, \rho) + \text{dist}(\rho, \beta) + \text{dist}(\beta, \delta)$
        $B \leftarrow \text{dist}(\alpha, \beta) + \text{dist}(\beta, \rho) + \text{dist}(\rho, \delta)$
        $C \leftarrow \text{dist}(\alpha, \beta) + \text{dist}(\beta, \delta) + \text{dist}(\delta, \rho)$
        $M \leftarrow \min(\{A, B, C\})$
        if $M \equiv A$ then
            $path \leftarrow path.\text{insert}(i, \rho)$
        else if $M \equiv B$ then
            $path \leftarrow path.\text{insert}(i + 1, \rho)$
        else
            $\{ M \equiv C \}$
            $path \leftarrow path.\text{insert}(i + 2, \rho)$
        end if
    end if
end if
BIBLIOGRAPHY


[BGR06] Thorsten Büiring, Jens Gerken, and Harald Reiterer. Usability of overview-supported zooming on small screens with regard to individual differences in spatial ability. In AVI ’06: Proceedings of the working conference on Advanced visual interfaces, pages 233–240, New York, NY, USA, 2006. ACM.


94


VITA
Graduate College
University of Nevada, Las Vegas

Mikołaj Michal Słomka

Degrees:
Bachelor of Science, Computer Science 2007
University of Nevada, Las Vegas

Thesis Title: Visual Occam: High Level Visualization and Design of Process Networks

Thesis Examination Committee:
Chairperson, Dr. Jan Pedersen, Ph.D.
Committee Member, Dr. Laxmi Gewali, Ph.D.
Committee Member, Dr. Evangelos Yfantis, Ph.D.
Graduate Faculty Representative, Dr. Aly Said, Ph.D.