Summary of the Disagreements Between Dr. Voyles and Dr. Stewart as Set Forth in Dr. Voyles' Declarations

(1) Whether there was a motivation to combine the teachings of the Gertz Reference and the Morrow Reference;

(2) Implications of the ~~The proper~~ definition of the term "primitive operations" as used in the '236 and '557 Patents;

(3) Whether or not the operation performed by a "control task" ~~is~~ Can be primitive;

(4) Whether or not Onika is a programming and execution environment; and

(5) ~~Whether configuration files are a type of code~~.
Whether or not ~~Does~~ the Onika VPE produce programs or "software code"

ROY-G-BIV CORPORATION
EXHIBIT 2017
ABB v ROY-G-BIV
TRIAL IPR2013-00062

ABB v RGB
Δπ EXHIBIT
RGB Ex. 2020
Deponent
IPR 2013-62; 282
Date Rptr.
WWW.DEPOBOOK.COM

Ph.D. Dissertation

A Visual Programming Environment for Real-Time Control Systems

Matthew Wayne Gertz

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in Electrical and Computer Engineering

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

November 22, 1994

dedicated to the memory of my grandfather

HARVEY H. GERTZ
1916-1985

who taught me that no matter how bad the weather is outside, the cows still need to be milked

Table of Contents

List of Figures

List of Code

List of Tables

List of Abbreviations

CFIG: Chimera Configuration File Reading Utility

CMU: Carnegie Mellon University

DH: Denavit-Hartenberg Robot Parameters

DOF: Degrees-of-Freedom (generally, the number of joints a robot has)

ENET: Network interface designed for Onika and Chimera

GUI: Graphical User Interface

HMI: Human-Machine Interface

IPL: Iconic Programming Language

INBOTH: Input port declared as both INCONST and INVAR

INCONST: Input Port Constant

INVAR: Input Port Variable

I/O: Input/Output

NDOF: Number of Degrees of Freedom

ONIFLOW: Onika Upper-Level Flow Element

ONIKON: Onika Upper-Level Icon

OUTBOTH: Output port declared as both OUTCONST and OUTVAR

OUTCONST: Output Port Constant

OUTVAR: Output Port Variable

PV:	Program Visualization
RTOS:	Real-Time Operating System
RTPU:	Real-Time Processing Unit
SIG:	Signal
SVAR:	State Variable
VL:	Visual Language
VP:	Visual Programming
VPE:	Visual Programming Environmant
VPL:	Visual Programming Language

Abstract

The development of real-time software for control systems is an expensive process, accounting for a significant portion of total application costs. This expense can be reduced by automating the software development procedure; for instance, by providing a software framework in which coded routines are small, portable, and reusable. However, to make the integration and control of these routines accessible to programmers of various expertise, and thus further reduce the amount of required resources, a user-friendly high-level programming environment designed for the creation of reusable real-time software is required. A programming interface of this type would not only allow for the rapid development of software, but would also considerably ease the process of debugging real-time code.

Much of the expense and tedium of software development is caused by the limitations of textual code. To use a textual language properly, the programmer must undergo expensive training. The deciphering, debugging, and use of real-time textual code is particularly time-consuming, especially when the code is cryptic, non-portable, and uncommented. The sequential nature of text can cause confusion when tracing a program flow through subroutines, recursion, and processes spawned in parallel. These "coding complexities" inevitably consume many resources, adding to the mounting costs associated with the system. This limits the amount of applications which may be developed for the system and, in turn, limits the system's usefulness.

In the past, researchers have created visual programming languages to address the problems of textual coding. However, these interfaces have been, in general, either very high-level and narrow in scope, or low-level and cryptic. Furthermore, these interfaces have not been designed with the specific requirements of real-time programming in mind. These requirements include the need to switch from one job to the next with minimal time loss, the need to modify the code of a job while it is executing, and the need to coordinate many jobs running in parallel.

In this dissertation, we present a multilevel/iconically-programmed visual programming environment called Onika. Its multiple interfaces directly connect with the underlying real-time operating system to coordinate the activities of several routines running in parallel. Each task on the real-time operating system is represented by an icon, which is manipulated by a mouse. These tasks are combined in a logical way to create jobs for the system to execute. The interface is able to switch from one job to the next quickly, in real-time, with minimal system delays. The user is also able to monitor and modify each routine running on the real-time operating system. Furthermore, a combination of routines created at one level can be saved as a reusable higher-level routine for others to use. Thus, routines at the higher levels become more specific, making programming accessible for naïve users, without diminishing the programming scope for more knowledgeable users working at the lower levels. Both levels of users are presented with an interface appropriate for their programming abilities and application requirements. We show that the grammar and syntax of the language supported by the interface is complete with respect to traditional programming languages, and we demonstrate via standardized user testing that this new method of programming is much faster and less error-prone than traditional methods.

Onika has been demonstrated at the NASA Langley Research Center and in several joint Sandia National Laboratory and Carnegie Mellon University distributed laboratory demonstrations, and is in regular use at several outside laboratory sites, including Wright Patterson Air Force Base and NIST.

Acknowledgments

If I were to create a list of all the many people who have given me encouragement and advice over the course of the research presented in this dissertation, the amount of paper required to print it might very well result in the demise of a small deciduous forest. Nevertheless, there are certain persons whose friendship and assistance simply cannot go unrecognized.

First, I would like to thank Dr. Pradeep Khosla, who has been my advisor at Carnegie Mellon University for both my M.S. and Ph.D. research. Pradeep's suggestions and advice helped to turn what started out largely as a brief flirtation on my part with hypermedia into a full-fledged visual programming environment, and his fervent dedication to making Onika highly visible to the research community as well as to the world-at-large has been extremely gratifying. His enthusiasm for this project has kept my spirits high, and I am very much grateful to him. I would also like to thank my committee members Dr. Daniel Siewiorek, Dr. Roy Maxion, and Dr. Kelli Willshire for their comments, suggestions, and advice, which was extremely helpful in making Onika a much more usable and robust system. I would particularly like to thank Roy for teaching me the basics of user testing, and for being available for answering my questions, for helping to guide my user testing, and for all of the interesting and humorous discussions we've shared over the past year. Additional and grateful thanks goes to Dr. David Banks of the Statistics Department at CMU for showing me how to determine the statistical significance of my subject test results.

My life would have been much more complicated without the aid of Debbie Scappatura, our secretary extraordinaire. Deb went well out of her way to help me publish and publicize my research, as well as to make certain that there was always at least one mushroom pizza delivered to our group meetings. I'd also like to thank Beth Cummins, Carol Patterson, and Pauletta Pan for their secretarial assistance over the past few years. A special "thank you" is reserved for Lynn Philibin and Elaine Lawrence of the graduate office, who wielded a mighty pair of scissors against the occasional maelstrom of red tape in which I occasionally found myself tangled. I am also most grateful to Anne Watzman of the School of Computer Science, whose work on my behalf has resulted in several mainstream publications about Onika, as well as a nomination for the 1995 Discover Award for Technical Innovation in Computer Software for Onika.

I've been fortunate enough over the years to have some really first-class officemates. My very good friend (and Best Man at my wedding) David Stewart really helped me find my way around when I first came to CMU, and his advice and assistance during our years of working together has been highly appreciated. I am extremely grateful to him for having developed the Chimera Real-Time Operating System (without which my research would have been much more difficult) and for dragging me out of my apartment and forcing me to have fun. I'd also like to thank Darin Ingimarson, my current programming partner and a heck of an engineer, who read my thesis cover-to-cover and made many helpful suggestions for revision. Wayne Carriker has been a great friend to me from the very first day I walked into my office, and his unique sense of humor, as thoroughly twisted as my own, has cheered me up on more than one occasion. I'd also like to thank my other officemates for their friendship over the years: Dean Hering, Wing Keung "Fred" Au, Marcel Bergerman, and Shin Teraji.

The Advanced Manipulators Laboratory has been a great place to perform my research, and I would like to thank its members, past and present, for their help, comments, and undying patience: Don Schmitz, Eric Hoffman, Raju Mattikali, Richard Volpe, Jin-Oh Kim, Nikolaous Papanikolopolous, Sue Hartman, Chris Paredis, Alexei Sacks, Fred Seiler, Brad Nelson, Richard Voyles, Dan Morrow, Anne Murray, Prithvi Rao, Brain Bankler, and Carol Hoover. It has truly been a pleasure to work with these highly talented people. I'd particularly like to thank Brad Nelson for his help with demonstrations, as well as helping determine how programming in Onika could be compared to conventional programming methods.

I would like to thank all of the organizations who sponsored me during my post-high school education, including the State of Michigan, the U. S. Navy, NASA Langley Research Center, Sandia National Laboratories, ARPA, and, most importantly, the Department of Electrical and Computer Engineering and The Robotics Institute at Carnegie Mellon University. I'd especially like to thank Jack Pennington, Mike Goode, Hal Aldridge, and Billy Doggett at NASA and Ray Harrigan at Sandia for all of their input and advice. I'd especially like to thank Philip Morris for his help and friendship while I was first experimenting with visual programming environments at NASA.

I could never give enough thanks to my grandparents, Warren and Anita Fuller, and my aunt, Kathy Fuller, for their love and assistance during the course of this research. It is no exaggeration to state that, without their help in my early years at CMU, I simply would have been unable to continue my Ph.D. studies. I have always been grateful to have been able to drive up to their lake in Michigan to dip my toes into the water during those times when I needed to get away from the world. I would also like to thank my mother, Diana Gertz, my father and stepmother, Wayne and Frances Gertz, my sister Barb, my brother Ryan, my brother Mike, his wife Monica, my niece Audrey, and my grandmother Edna Gertz for all of the love and encouragement that they've given me over the course of a lifetime.

But first and foremost, though, I'd like to thank my wonderful wife, Glenda, who makes it all worthwhile.

1. Introduction

1.1 Overview

This dissertation addresses the usability of real-time control systems with respect to reduction of resources required. Specifically, we address two issues: reduction of the time required for technology transfer, and reduction of the time and expenses required for training an individual to control a real-time control system. We present a *visual programming environment*, Onika, which can be used as a graphical front-end for a real-time operating system. The Onika interface includes different (but cooperative) levels of graphical programming for users of differing abilities, as well as an environment in which the user can completely control the executions of programs by the control system. Onika includes support for traditional programming structures such as loops, case statements, and breakpoints, while including new structures such as synchronization tags and aliases, as well as support for parallel subsystems running in a multiprocessor real-time environment. Additionally, programs designed for one manipulator can be executed in simulation or on another manipulator without changing the code of the program, keeping all details hidden from the user. User test results of Onika were analyzed to determine ways to enhance the system, and are presented in this dissertation to demonstrate the potential usability of this visual programming environment.

Our motivations for our research are presented in Section 1.2 of this chapter. The goals and contributions of this research are listed in Section 1.3. Finally, the organization of this dissertation is outlined in Section 1.4.

1.2 Motivation

The creation of code for real-time control systems (such as robotic manipulators)[1] is generally considered a difficult and time-consuming process, involving many hours of training, coding, and maintenance. Some of the specific problems associated with traditional coding methods include:

- *Expertise required*: Workers must be highly trained in programming, real-time operating systems, and control systems.

- *Programmers are not end-users*: Typically, the workers who create the code for machine systems are not the same as those who actually control the equipment, leading to the creation of code which might not be ideally suited for floor operations.

- *Textual nature of code*: Textual code is cryptic by nature, and can be poorly commented, leading to confusion on the part of another programmer irrespective of that person's expertise. Furthermore, the sequential nature of code is not at all suited for the coding of parallel subsystems. Textual code can also be quite difficult to debug, especially when parallel flows of control are involved.

- *Simulations*: Often, code for a physical control system must be rewritten entirely in order to execute it on a simulator, allowing more errors to be introduced into the code. If new code is then added to the simulator to make it operate in a new way, this code must then be ported back to the physical system code, potentially adding more errors.

- *Technology transfer*: Transfer and reuse of real-time application software is difficult and often seemingly impossible due to the incompatibility between hardware and systems software at different sites. This has meant that new technology developed at one site must be "reinvented" at other sites, if in fact it can be incorporated at all. Technology transfer, therefore, has been a very expensive endeavor.

The problems associated with traditional programming methods can create a serious drain on human resources; anecdotal evidence suggests that the creation of moderately complex real-time applications for control systems can take days or even weeks, with more overhead added if a new system needs to be brought on-line. To reclaim these lost and wasted resources, the nature of the programming process must be significantly altered. A system

1. Throughout this dissertation, we will use robotic manipulators as convenient examples of real-time systems. Nevertheless, the research presented in this dissertation is applicable to a wide variety of state variable-based real-time control systems.

more conducive to ease-of-programming and technology transfer would have the following attributes:

- *Programmable by end-users*: The end-user has the greatest knowledge as to how an application should proceed, and therefore should become involved in the programming processs. Furthermore, he or she should be able to make modifications to the program as required without needing to send the problem back to the low-level programmers.

- *Visually programmed*: Unlike traditional textual languages, visual programming could allow the user to see exactly what an application does, even when parallel flows are involved. Typos and spelling errors would be eliminated as well, and the construction of code could be done in a more intuitive manner, with much less training involved.

- *Transparent simulation*: The systems simulator should use the same code as the physical system, eliminating errors introduced by porting code.

- *Reusable code*: With the exception of the code which directly communicates to the physical system, all other code primitives would be generic and usable with any system, thus aiding both technology transfer and system set-up.

In the past, visual languages have been created to address specific points on this list; however, these languages have suffered from the narrowness of their scope, and have not been at all useful for complex systems such as real-time control systems. In the next section, we discuss the goal of this dissertation, which is to provide an environment capable of addressing all of the considerations given above. We also include a list of our contributions stemming from this dissertation.

1.3 Goals and Contributions

The goal of this dissertation is to provide a *visual programming environment* (VPE) in which software from various sites can quickly be assembled and used to control both local and remote hardware. This VPE is targeted towards both the control-level programmer and the application-level programmer; specifically, the code generated by the control-level programmer is used as the building-blocks of the language used by the application-level programmer. The VPE is defined as having the following characteristics:

- Code is created, simulated, and executed all within the same programming environment.

- The creation of code is done visually, with appropriate cues enabling the user to assemble code more quickly than possible with traditional textual methods.

- Code is modular, reusable, and can be shared with other sites.

- The execution of code can be fully monitored and controlled from within the environment, and adequate visual feedback is provided.

- Different programming interfaces are presented to users of differing expertise. The code generated by "lower-level" interfaces serves as the building blocks for "higher-level" code. Thus, the scope of high-level applications generated in the VPE are not limited, as is the case with other high-level programming interfaces, since the available "lexicon" of code is not static nor hard-coded into the VPE.

- The programming interfaces are customizable to the individual user.

- Communication with the control system is done over a network, and thus can be performed remotely.

- Any code written by conventional methods can be duplicated in the VPE (i.e., the available languages structures are complete).

We intend that the use of this VPE should significantly reduce the time involved in technology transfer and programming by several orders of magnitude, and should make the programming of real-time control systems accessible to persons who otherwise would not have the knowledge or training required to control such a system. Ultimately, this research will lead to the development of *distributed laboratories*, wherein users will be able to freely access real-time software available on the network, as well as be able to control systems remotely from their physical location, without needing to rewrite or modify any of the code involved.

In the course of the research leading to this dissertation, we have made several contributions to the field. First, we have developed Onika, an iconically-programmed VPE, which provides different (but cooperative) interfaces for both knowledgeable (lower-level) and inexperienced (upper-level) users. Onika is currently the only iconic interface for control systems which supports reusable and reconfigurable software [16][20]. Additionally, by using the Internet to communicate to the real-time operating system (RTOS) and to automatically download software for linking, an Onika user can control both local and remote hardware using both local and remote software, unlike other GUIs (graphical user interfaces).

To interact with reconfigurable software modules at the lower level, Onika uses a novel program visualization mechanism, in which the modules available in the libraries are displayed on-the-fly as icons having the appropriate input and output ports according to their underlying real-time port-based objects model. If the naming conventions of downloaded software differ from the user's convention, Onika provides a novel method for renaming state variables graphically, so that the software can be used immediately without needing to recompile code. Icons used in a configuration of modules are automatically connected, removing the need to manually connect I/O ports as is the rule in other GUIs.

While the upper level supports all of the language structures of typical textual languages, Onika's "language" and "syntax" can be further expanded by the user as required[21], which removes the "limited scope" problem found in other GUIs (which tend to be domain-specific). A background syntax checker ensures that all applications are complete and syntactically correct. As is the case with the lower level, Onika can control and synchronize parallel subsystems executing on multiple CPUs at the upper level. Furthermore, Onika-code generated for one real-time control system can also be run on another system (barring gross physical dissimilarities) or on a simulator with a single mouse-click. This unique functionality is made possible by the use a clever built-in aliasing mechanism. This cross-system ability to run code eliminates errors which might be introduced when porting code between (for instance) a simulator and a manipulator. Additionally, the subjects we tested were able learn to use Onika to program a robot in less than 20 minutes, regardless of any lack of experience with programming or robotics.

We have also performed subject tests on Onika to determine its effectiveness as a programming environment. This has provided us with first-ever benchmarks for rating the performance of a subject programming a manipulator, which are measured in seconds. (Using traditional textual coding, programming a manipulator would take many hours or days, and in any event would have been beyond the knowledge of most of our subjects). Furthermore, subjects which had no special training in programming and robotics were able to create simple programs to control the manipulator within minutes, something which would have otherwise been impossible for them to accomplish.

1.4 Organization of Thesis

We now present our outline for the remainder of this dissertation. In Chapter 2, we discuss previous work related to visual programming languages, graphical user interfaces, and user testing. In Chapter 3, we introduce the terminology associated with the reconfigurable software framework developed at Carnegie Mellon University, under which Onika operates. In Chapter 4, we give an overview of Onika's functionality, and demonstrate how Onika can be used to support a distributed laboratory. In Chapter 5, we discuss the specifics of the mechanisms developed to support the Onika visual programming environment. In Chapter 6, we present the results from our user testing of Onika. Finally, in Chapter 7, we summarize this dissertation, as well as list our contributions and suggest directions for future work in this area.

2. Related Work

2.1 Introduction

Our research into visual programming environments has been driven by the need to allow both naïve and knowledgeable users to create reusable and reconfigurable real-time code using port-based objects, as well as by the need to facilitate technology transfer. These goals led us to explore many research areas related to interface design and software assembly for control systems, including high-level and low-level visual programming environments, program visualization techniques, use of colors and other visual cues, reconfigurable and reusable software, user testing, and hypermedia techniques. This chapter discusses the previous work in these areas related to the research presented in this dissertation.

2.2 Textual Languages

Initially, the programming of robots was performed using textual languages such as Pascal and C. The code required to drive robots was very low-level, and very hard to create and maintain, requiring skills in both advanced programming and control theory. Textual robotic languages such as VAL II [63] or AL [46] were among the first languages developed specifically to address the problems associated with programming robots. These languages introduced "built-in" commands to operate the robot, eliminating (for instance) the need to develop code for motion primitives. While they were highly instrumental in making robots more productive, these textual languages suffer from several flaws. First, programs written in textual languages are difficult to read, not only due to the cryptic nature of textual code, but because code can be poorly commented, and also because parallel-running applications must be mapped serially. There is no convenient means (beyond low-level semaphores) for synchronizing two flows of code except by the multiple use of "cobegin-coend" blocks. The languages furthermore have no facility for using multiple

CPUs running under a real-time system, and any code generated for them is robot-specific. Additionally, their support for simulators is extremely limited, and in fact much code must be rewritten in order to be executed on a simulator. Most limiting, however, is the fact that their grammar is not easily expandable. New actions can only be used via external procedure calls, and thus there is a confusing lexical difference between using "built in" functions and "homemade" functions. The usable domain of these languages is limited to single-controller robots performing basic "pick-and-place" operations, and they are not general enough in scope for real-time control system applications. An expandable language is required, wherein all functions can be addressed in the same fashion, programs are portable and can be run seamlessly on other types of real-time control systems, and programs can be simulated without changing a single line of code.

Because of these inherent limitations in textual coding, languages have begun to be introduced which use icons as a way to program systems. The evolution of programming languages from textual languages to visual languages and the rationale for this change are presented in [8].

2.3 Visual Programming and Program Visualization

Much of the work in the field of visual language environments has been in the area of data-processing, database retrieval, and vision processing [5][6][50][47]. Visual language interfaces have also been used in AI[8], low-level C coding [36], animation [62] and, of course, operating systems [39]. Additionally, there are interfaces whose sole purpose is to design other interfaces [11][55].

Graphical interfaces which use some sort of *visual language* fall within one of two categories: *visual programming* (VP) or *program visualization* (PV), both of which are subclasses of *visual languages* [50]. The former category is comprised of interfaces which allow the user to create programs graphically, whereas the latter category is comprised of interfaces which convert previously-written conventional code to a viewable form. In the past, the domains of these two categories have not intersected; VP interfaces allow for the creation of programs from graphical elements representing specific static-code procedures, whereas PV interfaces permit graphical viewing of arbitrary (though finite in range) procedures within a program without any capability to change the program (as in ROBSIM [14]). However, in a visual programming environment which involves reusable software, the user must be able to manipulate pre-existing and dynamic arbitrary procedures in order to create programs. Thus, traditional visual languages, VP and PV, are not useful for a reconfigurable system when taken separately; the system in fact requires both. In

[50], Myers states:

> ...It is more accurate to use the term Visual Programming for systems that allow the program to be *created* using graphics, and Program Visualization for systems that use graphics *only* for illustrating programs after they have been created.

By such a definition, the system which we have described would technically be referred to as a VP system, since the "only" qualifier in the PV definition eliminates any other possibility. That is, a VP system can have some PV aspects, such as graphical debugging, and yet still be considered VP. However, it is unclear in our minds that such definitions were meant to apply to a system with extensive use of both PV and VP techniques, as would be needed in the ideal set-up for real-time control systems. We therefore anticipate a need for a new class of visual language which extensively incorporates both PV and VP techniques. Such a hybrid visual programming language would permit the graphical creation of programs (as in VP) from visualized pre-existing conventional code (as in PV). The pre-existing conventional code could be adjusted external to the system, and subsequent system sessions would then reflect these changes using PV techniques. The code could then be configured interactively using VP techniques.

2.4 Presentation of Graphics

Only a limited area of space is available on a graphic display, so that it becomes difficult to present abstract information in a graphical user interface; the designer must be careful to determine where abstraction is useful and where more information is required. Frey *et al* investigated this "big graphic—little screens" problem in [12], wherein they showed that the acceptable level of abstraction for given user tasks varies according to whether the task is a "thinking task" or a "doing task." "Thinking tasks" benefit from a higher level of abstraction, whereas "doing" tasks are likely to require much less abstraction. HI-VISUAL '89 [32] is an example of a highly abstract paradigm applied to the "thinking task" programming process. The user selects a variety of tools, such as calculators, files, pens. etc., from a "workspace environment" (which resembles a typical office). These elements are combined in flow graph notation to generate a program, which is then submitted to the "secretary" sitting at another desk on the screen to be executed. This clever interface is well-suited for an office environment, but would be unsuitable for (for instance) a laboratory environment. ISHeE [49] is an example of another very specific "thinking task" user interface (designed in this case for programming simulations of hepatic cell behavior) which would not be adaptable to the general control of real-time systems, although their method of specifying syntax via the shapes of the icons is excellent. In general, interfaces which are designed specifically for a certain set of circumstances are difficult to adapt to other circumstances, and in our research we have tried to keep our interfaces as general as possible. Thus, less thought and abstraction are required when building Onika programs, making the presentation of graphics in our interface "doing-task" oriented. We have also avoided the traditional "flow-chart" presentation of graphics, which is highly wasteful of screen space.

Beretta *et al* [2] point out that the designer is responsible for clarifying the meaning of an icon, defining an appropriate symbol, designing the internal representation, and testing the icon. If a visual programming interface is to be widely used, the icons and the environment must be meaningful to all potential users [61]; otherwise, the performance of the interface may be worse than that of textual systems [54]. However, people with different cultural backgrounds will react to colors, shapes, and pictures in vastly different ways. Icons appropriate for a person from (for instance) the Western Hemisphere may be viewed as comic, obscene, or even insulting to a person from (for instance) the Middle East [38][66]. We have eliminated the "guesswork" from the design of icons by making all of the features of the Onika icons user-definable. Colors and shapes of edges may be tailored to suit the user's tastes without affecting portability, and each icon's picture can be replaced by a user's preferred picture using a built-in painting program. Thus, the meaning of the icon and its symbol are designed by the user as best suits him or her, eliminating the need for testing the icon. This leaves the designer to worry about supporting a "universal" internal model for any given icon, which in any event is transparent to the user, and not have to worry about such things as misuse or poor juxtaposition of colors [3][33][48]. We have also added the capability for the user to display brief textual descriptions near each icon, which has added to the usability of the interface; this agrees with similar findings in [66].

MacLean *et al* propose *user-definable buttons* (essentially "macros") as a way of achieving "tailorable interfaces" which can suit the needs of "workers," "tinkerers," and "programmers," all of whom have different goals when using the system (and all of whom interact via a "handyman"). However, for real-time control systems, the type of programming varies greatly between the goal-oriented "workers" and the control-oriented "programmers," necessitating a different type of interface entirely for either level. Nevertheless, user-definable buttons are available at all levels of Onika to further customize the interface.

2.5 Modeling Visual Programming

Chang [5] has developed an algebra for visual languages relating the logical and physical characteristics of a data object with the processes that transform it. This algebra is adequate for data processing tasks (such as visualizing textual code), but is not applicable for high-level goal-oriented application for real-time control systems, where the "output" is generally not a desired transformation of a data object, but instead is a transitory physical action. In this case, a new "complete" high-level visual grammar is required which can fully replace typical textual languages such as VAL II [63] while still hiding the details from the user. Brown has shown in [4] that models for visual programming environments can be created that are "complete."

The pros and cons of using task models (i.e. goal-oriented) vs. engineering models (i.e. control oriented) when building user interfaces are discussed in [13]. Where high-level users are concerned, the task model is more appropriate; on the other hand, low-level users generally find an engineering model more functional. We have resolved this problem within Onika by providing both types of interface, with the programs developed in the lower-level used as the building blocks for the upper-level code.

2.6 Syntax and Grammar

Data input can be a tedious and repetitive chore. For instance, when moving a robot to a joint position, the user may have to enter several lengthy numbers to indicate joint positions, speed limitations, trajectory duration, etc. If the user is moving the robot to a particular point often, the entry of data for that point becomes particular annoying. To alleviate this, many VP interfaces introduce the concept of "targets", which contain frequently-used information needed by procedures. For instance, a Cartesian procedure requires a Cartesian target in order to run, and a joint procedure likewise requires a joint target. The set of procedure and target types is finite, and is generally hard-coded into the VP interface. If a new job/target combination is required, the VP interface itself must be rewritten. Such rewrites are expensive and can lead to poorly-written code. To solve this problem, Onika uses a mechanism by which new target and procedure types can be added to the VP interface without needed to rewrite the code. This mechanism allows for the modification of target information as well.

In order to be useful, the syntax of a visual language should be made readily apparent to the user. Many ways of defining the syntax of a language have been proposed [25][32][42]. "Natural language" (NL) is often touted as a mechanism for defining grammar and syntax, but it is not clear that the natural means of communication between humans is necessarily the best method for communicating between humans and machines [28].[1] Most methods proposed for defining grammar and syntax were primarily designed for data-processing and not real-time control; nevertheless, many of their elements which involve icon presentation were adapted for use in Onika. The icons in the BLOX Methodology proposed by Glinert [23] can reference lower-level structures, and their use can be constrained by color and edge information. His mechanism for presenting data is similar to that of the Onika application-level interface, except that we have incorporated properties such as aliasing, synchronization, and flow control to the icon elements in order to operate more than one control system at a time.

DACRON [41] used flowcharts for defining actions in its goal-oriented visual interface for direct knowledge acquisition. Each "icon" had exactly the same appearance as any other, no matter whether it is a functional element or a structural element. Large amounts of textual information were essential to this interface in order to make any program readable, and each icon with its accompanying text took up a premium of screen space. Programs created using DACRON would not be easily understood by persons who did not speak English, as the icons themselves do not provide any clues to their meanings. Nevertheless, their ISD design philosophy [42] of analysis, theory formation, system design and implementation, and usability testing is sound, and we have used such an approach when developing Onika.

Lingraphica™, a Robot Programming Language which uses the DeVAR environment, divides its iconic lexicon into six categories: *actors*, *actions*, *placements*, *modifiers*, *things*, and *other*. Its icons have both a visual component and a textual component. These icons are assembled in a storyboard format in order to create a program [37]. While excellent in its domain, this package lacks several features necessary for truly reusable code for reconfigurable systems. There are no clear visual clues in this system as to the syntax of the icons, and success in assembling a program relies largely on the user's knowledge of English syntax. There is no function mentioned which can turn a newly-created program into a "macro" for repeated use, and programs are limited to single-flow threads. All commands are very high-level, and there is no graphical means for defining low-level details of actions. At the other extreme, Robotica, a good example of a well-received robot interface, is math-oriented and tailored to be of most use to engineers, making it too low-level

1. In any event, "natural" is defined by the language and culture of the user, since the structures of human languages can vary greatly, and therefore NL would not be suitable for any programming environment that might have to span more than one culture.

for non-technically-oriented users [56]. We have avoided these scope problems in Onika by providing interfaces tailored to different types of users.

Negative feedback (wherein the user is not only told that two icons are incorrectly connected, but the reason why such a connection is incorrect) is put forth in [34] as a mechanism for eliminating errors when connecting icons in a multiple flow configuration. The engineering level of Onika takes this one step further by doing all of the connections itself automatically as icons are inserted into the configuration, thus eliminating the possibility that the ports of different icons will be connected incorrectly.

2.7 Usability

One of the first graphical programming environments developed for general systems was Glinert and Tanimoto's PICT [24], a flow-oriented interface for lower-level programming. One of PICT's main novelties lay in the fact that its designers were concerned that the user should *enjoy* using the interface; i.e. not looking at programming as some tedious chore.[1] Malone [43] has shown that users respond more positively to interfaces (and learning situations in general) if the task is perceived to be "fun," and we have endeavoured to incorporate these ideas into our own Onika interface.

Glinert and Tanimoto also recognized differences in cognitive style, noting that users who, by their description, were field-dependent were more likely to benefit from PICT [24]. (Field-dependent persons tend to rely on examples and external referents to solve problems, whereas field-independent persons tend to adopt a more experimental approach [7][65].) Our own research and development environment is much more oriented to the field-independent cognitive style, and we show later in this dissertation that Onika is more intuitive to persons with this type of cognitive style.

2.8 Modularity and Hypermedia

Structured decomposition of higher-level tasks into lower-level tasks has recently been heavily researched issue, resulting in architectures such as NASREM [1] and the Reconfigurable Software Framework for Sensor-Based Systems [57]. Onika resides at the top level of the latter of these architectures, which has an advantage over NASREM in being more general for all sensor-based systems.

Truly modular software should be generic enough to be shared among various laborato-

1. The user-testing questionnaire for this interface included such questions as "In general, how much fun did you find it to use PICT?" and "How thrilled would you be if you were asked to participate in another PICT session?"

ries. All previous interfaces designed for control-systems have been limited in that code is generally not portable between systems, and must be rewritten during technology transfer. The ideal interface would be able to "go out" across a network and retrieve code from other sites, perhaps using the well-known Dexter Hypertext Reference Model [26][27], and seamlessly incorporate this code into the system. The actual physical location of the code would be hidden from the user, though retrievable if needed. The model could be expanded to access remote hardware as well as software, allowing users access to equipment otherwise out of their means of reach. Onika currently implements the Dexter Model to a degree, and initial work has been performed to expand Onika's capabilities to perform remote software access "on-the-fly" during a programming session.

2.9 Environmental Interfaces

Environmental interfaces are those in which the visual presentation of information exactly matches that of the workcell, and programs are specified "virtually." An example of this is Sandia National Laboratory's excellent GISC interface [29]. GISC is a "program-by-example" environmental interface wherein the user indicates the desired end-state of the system, and the planning algorithms handle the generation of the necessary robot motions. Direct operator control is rarely allowed except when circumstances dictate, and then only by highly skilled operators. GISC is an extremely effective strategy for worksite robotics, but is not appropriate for use in a research environment, where the user needs to have complete control over the manipulator's actions down to the lowest level, and where many "canned" applications might need to be used sequentially. GISC excels in detecting error conditions (e.g. the manipulator crashing into an obstacle) by simulation of the plan before run-time. We have incorporated a similar mechanism which allows us to simulate code before running it, and work is proceeding on an environmental interface which will generate Onika code.

2.10 Summary

In this chapter, we reviewed the previous work in the area of visual programming environments, starting with the transition from traditional textual languages to graphical programming. We discussed the relationship between visual programming and program visualization techniques, the presentation of graphics, and previously used models for visual programming computation, grammar, and syntax. We also discussed interface usability, program modularity, technology transfer, and environmental interfaces.

3. Terminology

3.1 Introduction

The Onika Visual Programming Environment operates within the context of a reconfigurable software framework (shown in Figure 1) developed at Carnegie Mellon University. In this chapter, we review the terminology associated with our framework as it pertains to Onika. Those wishing for a more detailed description of the software framework for reconfigurable multisensor systems and of the real-time operating system which supports its lower levels should refer to [57] and [60].

In Section 3.2, we discuss the terminology relevant to Onika and its place within the reconfigurable software framework. We then summarize this chapter in Section 3.3.

3.2 Framework Terminology

3.2.1 Control Modules

The lowest level of code which Onika works with in the reconfigurable software framework is the *control module.* A control module is an instance of a class of port-based objects, having zero or more inputs and zero or more outputs. (Details of port-based objects are given in [60].) Each control module is coded textually in a conventional manner (i.e. using C). The control modules have a fixed format, which allows one control module to be easily swapped for another without worry of violating intermodule communication protocol. These formats include procedures which the module executes when changing states (i.e. from inactive to active, from cycling to inactive, etc.).

Besides being responsible for updating system state variables, the control module must also be able to signal the graphical user interface whenever appropriate. For instance, if a module is designed to generate a trajectory over a certain duration for a certain job (as defined in Section 3.2.5), then it must signal the configuration controller before turning

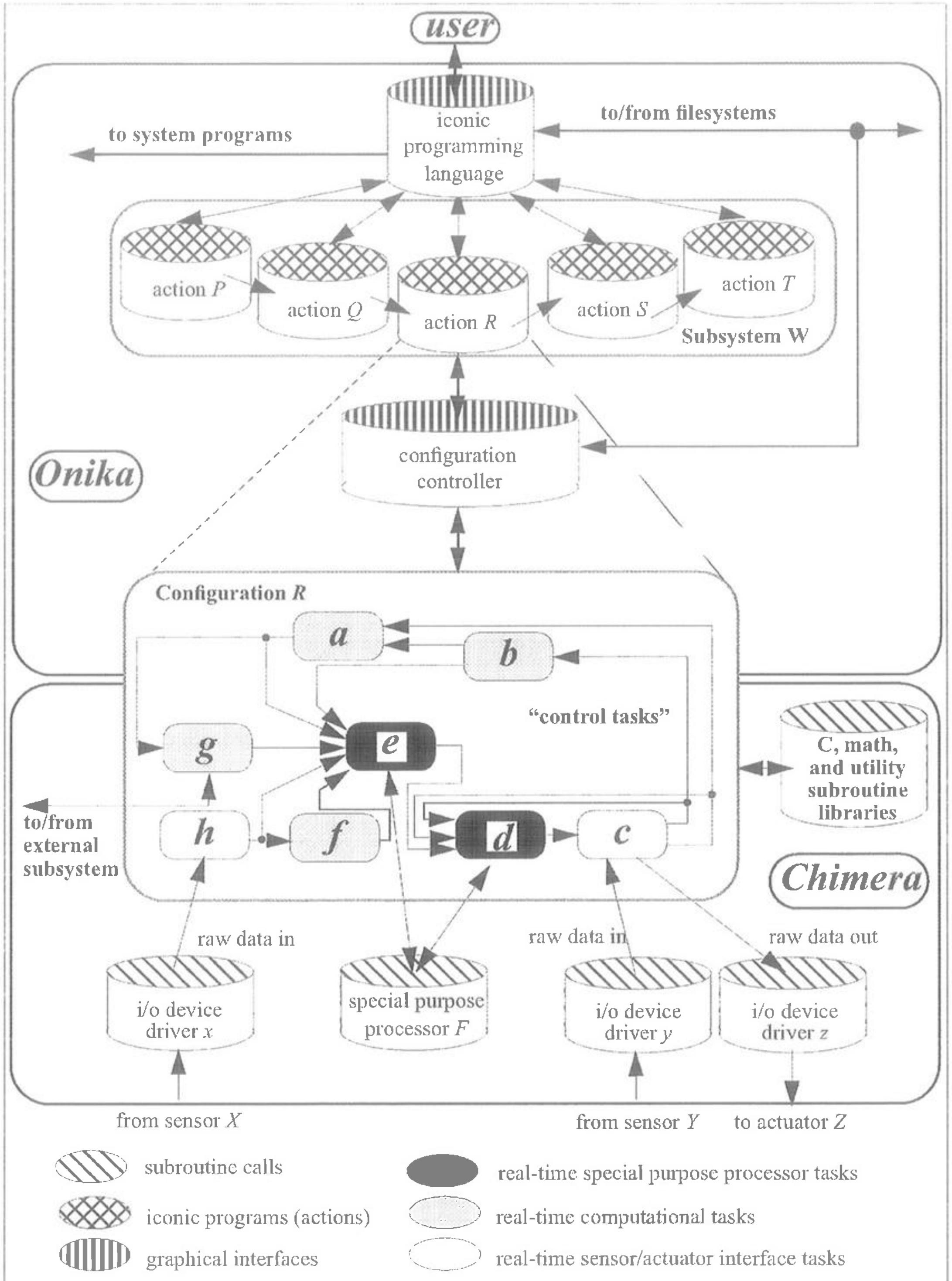


Figure 1: The Reconfigurable Software Framework developed at Carnegie Mellon University.

Page 42 of 198

ABBINC_0125798

itself off. In this way, the configuration controller will know to proceed to any subsequent job, and will be able to differentiate between the case where a module turns itself because the job is finished versus the cases where a module's deactivation is passive or where the module has gone into an error state. Control modules which signal the end of a job by their deactivation are referred to as *trigger modules*. Control modules for which the deactivation does not affect the job in any way are referred to as *passive modules*. Both types of modules are non-intersecting subsets of the general class of control modules. Modules which are neither passive modules nor trigger modules do not, as a rule, deactivate themselves.

Certain modules require user input (e.g. a trajectory-generating module which requires the endpoint of the desired trajectory), or communicate with other external subsystems (e.g. a module which sends data to an external real-time data display). User input can be received via the configuration controller, but external subsystem communication is done directly from the module to the appropriate subsystem without going through Onika.

While modules can and do directly communicate to the configuration controller via the connection between the RTOS and Onika, the configuration controller itself communicates to the modules via their *control tasks*. The next section discusses the relationship between modules and tasks.

3.2.2 Control Tasks

A *control task* is formed from the union of exactly one control module and a file containing various task parameters, such as task frequency, names of inputs and outputs, and other task-specific information. The relationship between the set of tasks and set of modules is not one-to-one; a module may be referenced by several tasks, as shown in Figure 2. By changing the parameters in the task file, the user can change the operation of the task without needing to recompile or relink code. Certain parameters can even be changed while the task is operating. Because each task refers to exactly one module, the terms *task* and *module* tend to be used interchangeably. Within Onika, a task is represented as a box with inputs on one side, outputs on the other, and its name, rate (if periodic or synchronous), and state displayed in the middle.

Since a module may not have more than one instantiation at any one time within a given control subsystem on the RTOS (as defined in Section 3.2.9), only one task which refers to a specific module may be included in a subsystem. Consequently, the terms *module* and *task* become interchangeable within the context of a subsystem. A *trigger task* is a task associated with a trigger module, and a *passive task* is a task associated with a passive

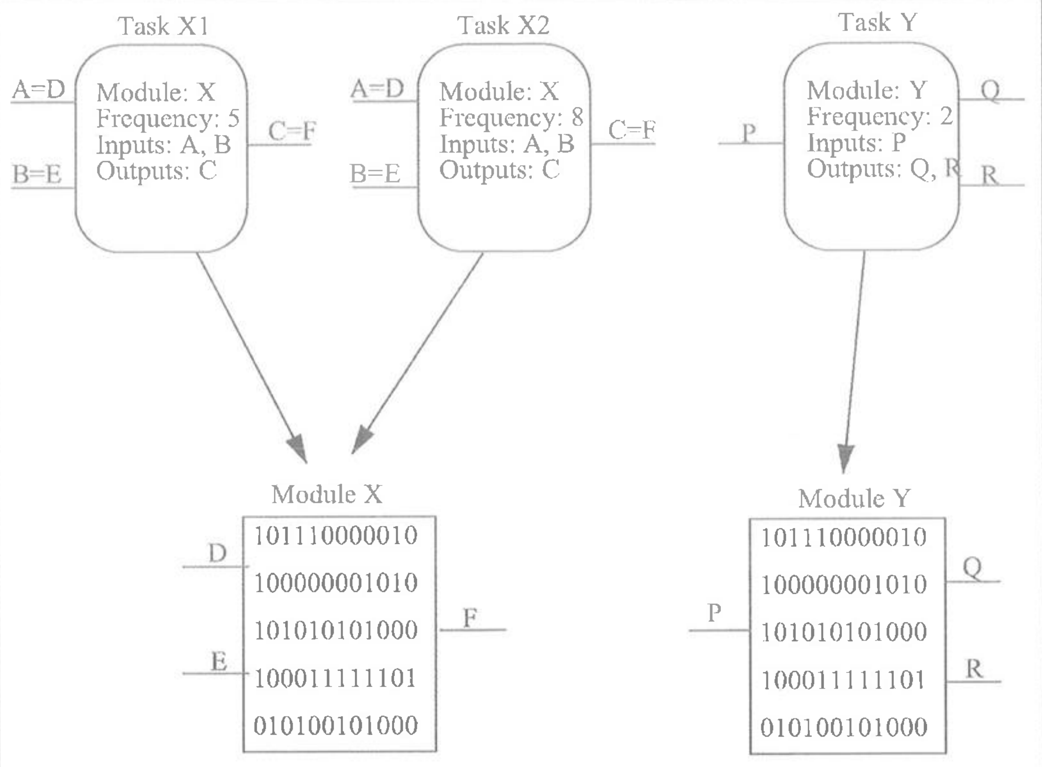


Figure 2: The relationship between tasks and modules. Note that the task can "alias' the names of the internal variables of the modules to more preferred names.

module. As before, these classes are non-intersecting and are subsets of the general class of control tasks.

Tasks are stored as object files and parameter files on a filesystem. The next section discusses these collections of available tasks in more detail.

3.2.3 Task Libraries

A *task library* is simply a directory which contains task files and the control module object code. The directory may be on the same filesystem as Onika, or may be on a remote filesystem (in such a case, Onika retrieves the remote files automatically using *ftp*). There can be multiple task libraries opened for concurrent use within Onika. Onika uses an environmental variable set by the user to determine the location of the desired task libraries. The tasks within these libraries are parsed, and icons are generated "on the fly" for each valid task. The icons of the tasks are then displayed in an appropriate window, as shown in Figure 3 on page 28. By interacting with these icons using a mouse, the user can modify task parameters, view the module code, or can spawn them on the RTOS. This latter ability is discussed further in the next section.

3.2.4 Configurations

A *configuration* is a set of tasks taken from a from a library which, when assembled together and activated, will perform a specific job, such as "move to *x*." This assembly is performed simply by selecting the appropriate tasks from the library and inserting them into a configuration workspace. For instance, a robot task, gravity compensation task, differentiator task, and trajectory task might be assembled to form a joint motion configuration, as shown in Figure 5 on page 29. In Onika, the common I/O ports shared between tasks are automatically connected graphically. This is done by comparing the names of the I/O ports as the icons are placed into configurations. These configurations can be saved for future use.

If Onika is connected to the RTOS, then the configuration controller will command the RTOS to spawn a task on a user-specified RTPU as it is placed into the configuration. The tasks in the configuration can then be completely controlled by the user via the configuration controller. Additionally, their parameters can be changed as is done in the task library.

Certain configurations are invalid; for example, a configuration that has two tasks which generate the same output variable would be liable to race conditions, and hence is illegal. Onika's syntax procedures prohibit such illegal configurations from being created.

Each configuration can be viewed as a single object which performs some *job*. These jobs are discussed further in the next section.

3.2.5 Jobs

A *job* is a high-level port-based object, which refers to a specific configuration of control tasks. Whereas lower-level objects have definite input and output ports based on state variables, these high-level port-based objects merely have ports to receive user-specified input. For example, a job which performs motion in joint space requires data specifying the endpoint of the trajectory. The instantiation of this target data within the software framework is discussed in Section 3.2.6. Certain jobs may not require any such target input, but may be self-contained.

In Onika, jobs are rendered as mnemonic pictures. Edges of a particular shape and color which indicate the type of job are appended and prepended to each picture. When assembling a subsystem (as discussed in Section 3.2.9) from these jobs, these edges dictate the syntax of the high-level language, indicating which pictures can follow any given picture in the subsystem.

Jobs have only three major properties: a link to the configuration which they represent, a description, and a *class* which specifies which edges are assigned to it, and hence the type of job (e.g. complete job, Cartesian job requiring a target, joint job requiring a target, etc.) Those jobs which do not belong to the "complete" class require some additional information to perform their operations. The type of additional information (if any) that a job requires is generally determined from the target task within the configuration which the job represents. For instance, if the configuration's trigger task is a Cartesian trajectory generator which requires a Cartesian endpoint, then the job linked to that configuration will be in the "Cartesian job requiring a target" class. In the next section, we discuss how these targets are implemented.

3.2.6 Targets

A *target*[1] is also a high-level port-based object, which supplies the data required by a job (or, more accurately, the data required by the trigger task of the configuration linked to the job). The graphical aspects of a target are similar to those of a job, with which they are combined to form a complete action. The name "target" is perhaps misleading; targets need not be (for example) locations in space, but can be as varied as a numeric index to use in an internal case statement, or a filename to which data can be written.

In Onika, targets are represented as mnemonic pictures which have edges and classes which are complimentary to those of job pictures. For instance, the right edge of a Cartesian job's picture is the same color as the left edge of a Cartesian target's picture, and the edges interlock in jigsaw-puzzle fashion. When a job is joined with an appropriate target, the resulting combination is a completely-defined *action* (as defined in Section 3.2.8). Graphically, both jobs and targets are referred to as *onikons*[2], and are stored in *job dictionaries*, which are discussed in the next section.

3.2.7 Job Dictionaries

Job dictionaries are similar to the task libraries presented in Section 3.2.3, except that the store the relevant files for jobs and targets (referred to collectively as *onikons*) rather than for control tasks. As with task libraries, the directory of the job dictionary may be on the same filesystem as Onika, or may be on a remote filesystem. There can be multiple job

1. Also referred to as an *object* in our previous papers; we use the term *target* in this thesis to avoid confusion with the term *port-based object*.
2. *onikon* = "Onika icon."

dictionaries opened for concurrent use within Onika. Onika uses the same environmental variable set by the user to determine the location of the desired task libraries to also determine the location of the job dictionaries. The jobs and targets within these dictionaries are parsed, and previously-stored mnemonic pictures are loaded in for each "onikon." The "onikons" are then displayed in an appropriate window, as shown in Figure 12 on page 36. By interacting with these icons using a mouse, the user can assemble the job and target "onikons" into actions which can be executed on a real-time control system. These actions are further defined in the next section.

3.2.8 Actions

An *action* is the syntactically-correct combination of a job and a target. Certain jobs are self-contained, requiring no target, and therefore are *de facto* actions themselves. For instance, a job which causes a robot to move to the specifications of a 6 DOF trackball does not require a target (as the "endpoint" of the trajectory is continually being updated by the trackball rather than from some fixed data scrap), and hence is a complete action in and of itself. A job which causes the robot to move to some specific location in joint space, however, is not a complete action unless joined with a target which specifies the location to where the robot should be moved.

Actions can be viewed as "steps" towards achieving some goal; e.g., "move to the peg," close the gripper," "follow the input of the joystick," etc. Actions are assembled in sequential order in order to create goal-oriented control subsystems, as discussed in the next section.

3.2.9 Control Subsystems

A *control subsystem* is a collection of actions which are executed one at a time, and can be assembled by a user. It has a definite beginning and end (represented in Onika by start- and stop-light pictures, respectively). The subsystem can be thought of as a goal-oriented program *flow* specific to a single real-time control system; for instance, "place all of the pegs in the appropriate holes," or "retrieve and place all of the parts necessary for assembly of machine part P-2341."

Subsystems may be "iconified" and stored as an "onikon" in job dictionaries for use in other subsystems, allowing for the creation of routines of even higher levels. These iconified subsystems have graphical attributes identical to those of a job, so that an Onika user may not even be aware that the "job" he or she is adding to a subsystem in fact represents a collection of actions rather than a single job; such differences are transparent to the user unless the user requests otherwise. If the subsystem is missing a target, then the resulting iconified subsystem is implemented and displayed as a "job requiring a target." Otherwise, the "iconified" subsystem is treated as a "complete job" (and hence a complete action).

Within Onika, the flow of a subsystem can be completely controlled through the use of breakpoints, top- and bottom-test loops, and case statements. When one or more subsystems operate in parallel, the resulting construct is referred to as an *application*, as discussed in the next section.

3.2.10 Applications

An *application* is one or more *subsystems* executing in parallel towards some common goal. For instance, one subsystem might cause a robot to place parts in certain locations for another robot to pick up, while a second subsystem might cause the second robot to pick these parts and assemble them in a certain order. Onika has a mechanism whereby actions in two different subsystems can be *synchronized*; that is, guaranteed to begin execution at the same time, freeing the user from needing to "tweak" action durations or add "pauses" in an attempt to cause to both actions to be reached at a certain time.

In the next section, we summarize the terminology presented in this section.

3.3 Summary

In this chapter, we reviewed the terminology associated with the reconfigurable real-time control software framework as it pertains to Onika, and discussed how Onika implements this framework.

The lowest level of the framework with which Onika interacts is the *control module*, which is simply a real-time port-based object. The association of a control module with a parameters file results in the creation of a *control task*, which performs some operation such as "read data from trackball" or "perform forward kinematics."

Control tasks are stored in *task libraries*, which Onika accesses to create icons on-the-fly for each task. It then displays the icons in an appropriate window. The control tasks, which execute in parallel, are combined (using their icons) to form *configurations*, which perform some real-time control function (such as "move to *x*"). A picture can be assigned to a configuration, allowing it to be used at a higher-level as a *job*.

Certain jobs require targets (for instance, the *x* in "move to *x*"). These targets also have pictures assigned to them. The target and job pictures are referred to collectively as *onikons*. Each "onikon" has color- and shape-coded edges to indicate syntax. "Onikons" are stored in a *job dictionary*, from which jobs and targets can be combined into *actions*.

An action is made up of either a job which does not require a target, or from a syntactically-valid job-target combination. Actions are assembled sequentially to form a *control subsystem* (also called a *flow*). Subsystems perform some goal-oriented function, and can themselves can be assigned “onikons” so that they can be used in other subsystems transparently. When one or more subsystems are operating in parallel towards some common goal, the resulting structure is referred to as an *application*.

4. Overview of Onika

4.1 Introduction

We now describe our *visual programming environment* (VPE), Onika, from the user's viewpoint. In this chapter, we present the various interfaces presented to the user, and show that the language is "complete" with respect to conventional textual robotic languages. We also describe a complete Onika session for both levels of the interface, as well as introduce the concept of *distributed laboratories*, which Onika supports in order to access both hardware and software resources remotely. Our intention is to give a high-level overview of Onika's functionality and uses. Specific details as to the implementation of any Onika capability are given in Chapter 5.

The Onika visual programming environment has proceeded through several stages of development during the course of this research. The original concept for a high-level iconic programming language (IPL) evolved during the summer of 1991, in response to the large amount of training required to operate laboratory manipulators. We developed a prototype upper-level interface of this type, called "Bookworm," on a Macintosh IIfx during an internship at NASA Langley's Automation Technology Branch in the fall of 1991. Bookworm was able to generate AL code for NASA's ROBSIM simulator's use in operating a simulated Laboratory Telerobotic Manipulator. However, except for target icons, all "jobs" in Bookworm were hard-coded, with no easy way to expand them. Essentially, Bookworm was a front-end interface for AL.

Development of the Onika lower-level interface began in January 1992. This work was done on both the SunView and Macintosh platforms. It was decided that this level would be developed simultaneously with both the evolving Reconfigurable Software Framework and Chimera 3.0. Prototype work was completed in the summer of 1992, at which time the SunView platform was abandoned in favor of the X platform (using XView library routines to facilitate coding). Onika 1.0, comprising only the lower-level functionality of the present-day Onika, was interfaced with the Chimera 3.0 Real-Time Operating System in November of 1992 using a generic socket library we developed for this purpose. During Onika 1.0's development, the upper-level interface was designed on a Macintosh IIci. This level was ported to X in the spring of 1993, and made functional in Onika 1.1 in the fall of 1993. Error handling was added to this version as well.

Support for conditionals and parallel structures was added to version 1.2 (released October 1993). Onika 1.2 also took full advantage of the Chimera *cfig()* libraries to support ASCII parameter files, rather than binary files. Version 1.2 was considered alpha-test, and was demonstrated at various sites around the country.

Version 1.3, the beta-test version, was released in November 1993, and included a mechanism for backwards-compatibility of code generated by Onika 1.2 or later. Version 1.3 went through many changes through May 1994 (version 1.38), including the addition of synchronization tags, top-test and bottom-test loops, breakpoints, as well as a much faster execution loop. The current version, Onika 1.41, was released September 6, and includes changes suggested by our user-testing.

We begin the overview presented in this chapter in Section 4.2 by introducing the *lower* (or *engineer's*) level of Onika, wherein reconfigurable modules are combined to form upper-level *jobs*. We then demonstrate the use of these jobs (along with their accompanying *targets*) to create *applications* in the upper-level of Onika in Section 4.3. In Section 4.4, we show that the grammar and syntax of the upper level interface is complete with respect to the textual robotic language Extended AL described in [46]. Demonstrations of Onika's use in hardware and software technology transfer to support a *distributed laboratory* are given in Section 4.5. Finally, in Section 4.6, we summarize the Onika interface and its uses.

4.2 Lower Level Interface

In the lower level interface of Onika, upper level jobs may be created by combining certain modified port-based objects called "tasks" into a format resembling that of control-block diagrams. Knowledge of textual coding is not required, but merely a good working knowledge of control theory. These configurations of tasks can be executed and fully controlled from the lower level interface. In this section, we give an overview of the lower level interface.

```
MODULE                rvel_gen
DESC                  reference velocity generator
SVARALIAS             Z_Y=Q_REF
SVARALIAS             Z^_Y=Q^_REF
INCONST               NDOF
OUTCONST              none
INVAR                 Z_Y
OUTVAR                Z^_Y
TASKTYPE              periodic
FREQ                  250

EOF
```

Code 1: An example parameter file for a task. Onika uses these files to create icons "on the fly" for each task.

4.2.1 Combining task routines

The basic unit of combination at the lower level is the *task*. As mentioned in Section 3.2.2, a task is a *modified* control module. The module code by which the tasks process with their input values is written entirely in text. The tasks themselves, however, are represented by a single block-form icon having a certain number of input and output pins. The mechanism by which the task performs its function is hidden from the lower level user.

A parameter file is associated with each task's module (shown in Code 1). This parameter file completely describes the task. When Onika is executed, it loads in all available task parameter files on the system. These tasks may be located on the local file system, or may be on remote file systems, in which case Onika will use *ftp* to retrieve and link the modules. It then creates icons on the fly for each task from information in the file. These icons are presented to the user in an area known as the *task library*, shown in Figure 3. To create a job by combining tasks, desired tasks are selected on the lexicon, and a copy is then be placed in the combination area. This combination area is called the *job canvas*. The specific rules for placing tasks on the canvas are discussed in Section 4.2.2.

When a task is placed on the canvas, it is rendered at the point where the user lets up on the mouse button (as shown in Figure 4). Onika then checks the pins of the new tasks and determines whether each has a similar variable name to other pins on the canvas. If so,

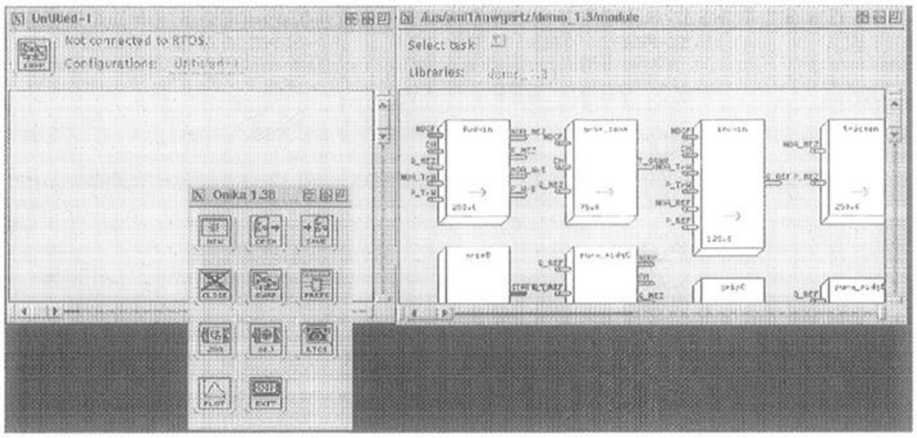


Figure 3: Clockwise from the top left: The Onika configuration canvas, the Onika task library, and the Onika control panel. Tasks are presented to the user in the library; a pop-up menu helps the user locate tasks in libraries containing many tasks. There is no limit on the number of open libraries within Onika.

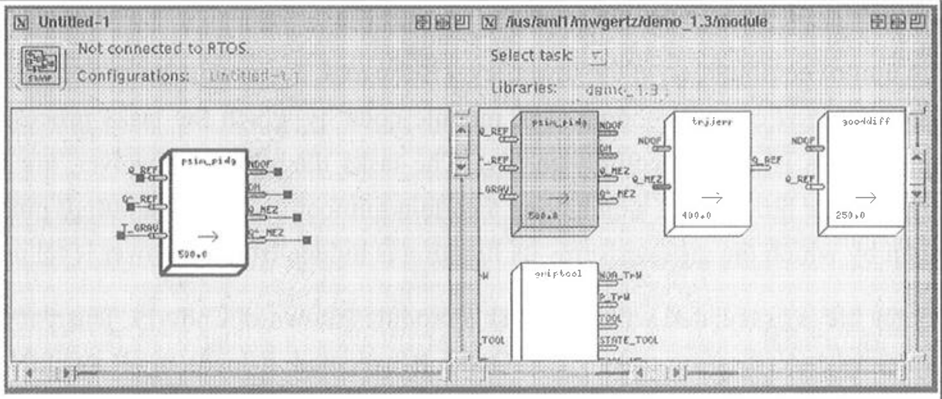


Figure 4: The user has placed a PUMA simulator task onto the canvas. Tasks which would conflict with its operations are "greyed-out" in the library, to prevent the user from placing them.

then these pins are graphically connected to each other, to illustrate to the user that these tasks are now connected in the supporting real-time operating system (Figure 5).

Onika can be actively connected with the real-time operating system. In such a case, as each task is dragged to the job canvas, it is spawned on the supporting RTOS. The user can toggle the state of activity of the task, can move the task's icon around on the canvas without affecting the system otherwise, and can delete (and replace) the task. Furthermore, a

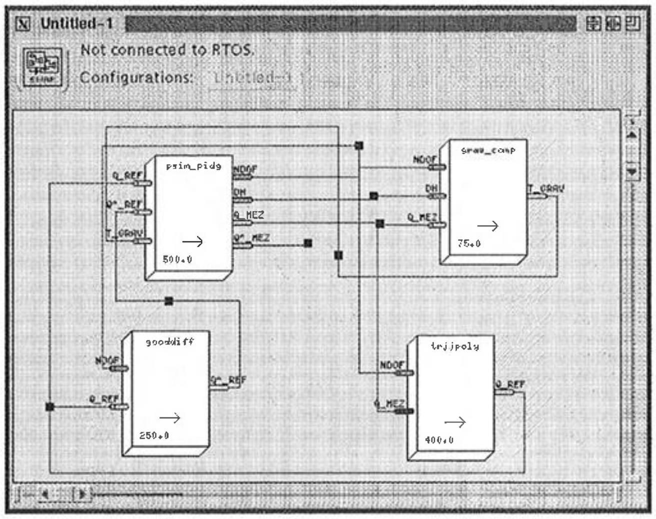


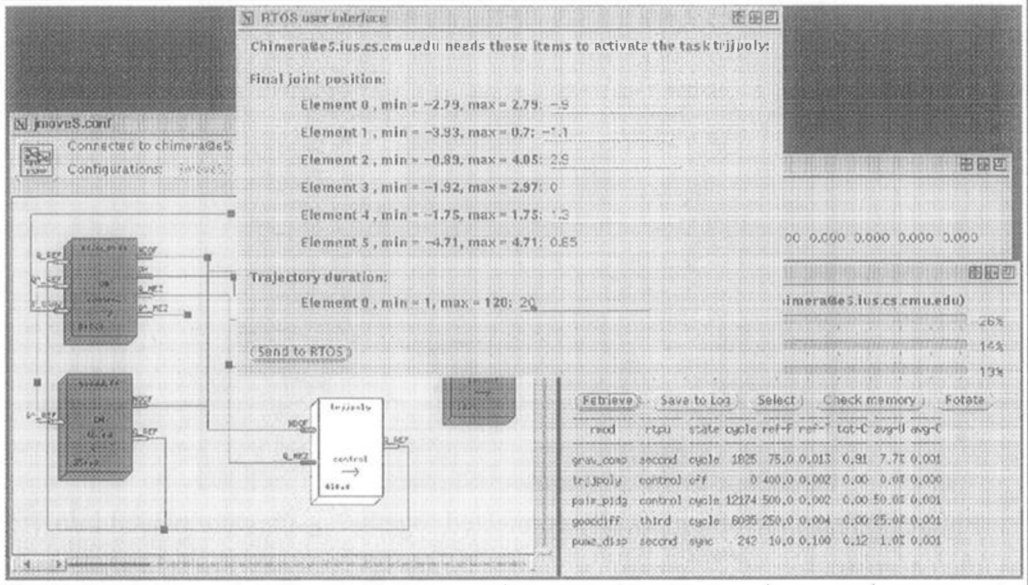
Figure 5: Tasks are automatically connected together as they are placed into the configuration.

combination of tasks on the canvas can be saved at any point for later recall. Multiple configurations may be open and executing simultaneously during an Onika session.

If Onika is connected to a RTOS, tasks are automatically spawned on this system when placed on the canvas. (If the connection to the RTOS is performed mid-session, then Onika will instruct the RTOS to spawn any existing tasks on the canvas if they are not already spawned within the RTOS.) If multiple CPUs exist, the user will be prompted as to the CPU upon which the task should be spawned. Once a task has been spawned, it can be "cycled" (i.e. turned on) by a simple mouse-click upon the task's icon. A task may be deactivated in the same manner, or cleared if it goes into error. Selected tasks may be deleted from the canvas using the "delete" key. Deleted tasks are "killed" on the associated RTOS.

Certain tasks (in general, trigger tasks) require additional input before they can cycle. For instance, a trajectory task requires an endpoint and a duration. These tasks are written to send a request to the user when the latter tries to activate them. Onika intercepts these requests, formats them, and presents them to the user in a window, as shown in Figure 6. The user enters the necessary information and sends it off to the RTOS. Onika will check to make certain that the entered values are within established boundaries before sending the information back to the RTOS, and will prompt the user to re-enter the information if it is outside the boundaries.

A variety of feedback is available to the user while connected to the RTOS. The status



RTOS user interface

Chimera@e5.ius.cs.cmu.edu needs these items to activate the task trjjpoly:

Final joint position:

Element 0 , min = -2.79, max = 2.79: -.9

Element 1 , min = -3.93, max = 0.7: -1.1

Element 2 , min = -0.89, max = 4.05: 2.9

Element 3 , min = -1.92, max = 2.97: 0

Element 4 , min = -1.75, max = 1.75: 1.3

Element 5 , min = -4.71, max = 4.71: 0.65

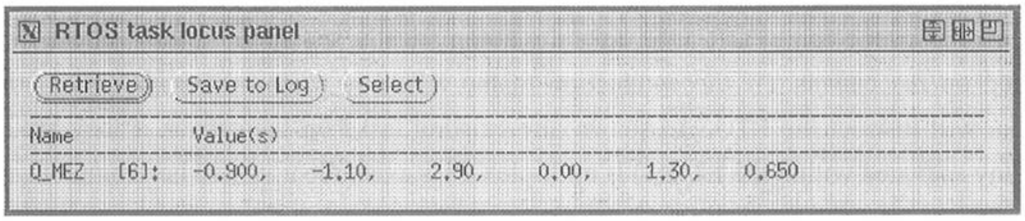
Trajectory duration:

Element 0 , min = 1, max = 120: 20

(Send to RTOS)

```
imera@e5.ius.cs.cmu.edu)
                                   26%
                                   14%
                                   13%

   rmod      rtpu    state cycle  ref-F ref-T tot-C avg-U  avg-C
grav_comp  second   cycle  1825  75.0 0.013  0.91  7.7% 0.001
trjjpoly   control  off       0 400.0 0.002  0.00  0.0% 0.000
pair_pidg  control  cycle 12174 500.0 0.002  0.00 50.0% 0.001
gooddiff   third    cycle  6095 250.0 0.004  0.00 25.0% 0.001
puma_disp  second   sync    242  10.0 0.100  0.12  1.0% 0.001
```

Figure 6: The user has "cycled" most of the tasks. The final task is a trajectory task, and requires an endpoint and a duration to begin execution.



RTOS task locus panel

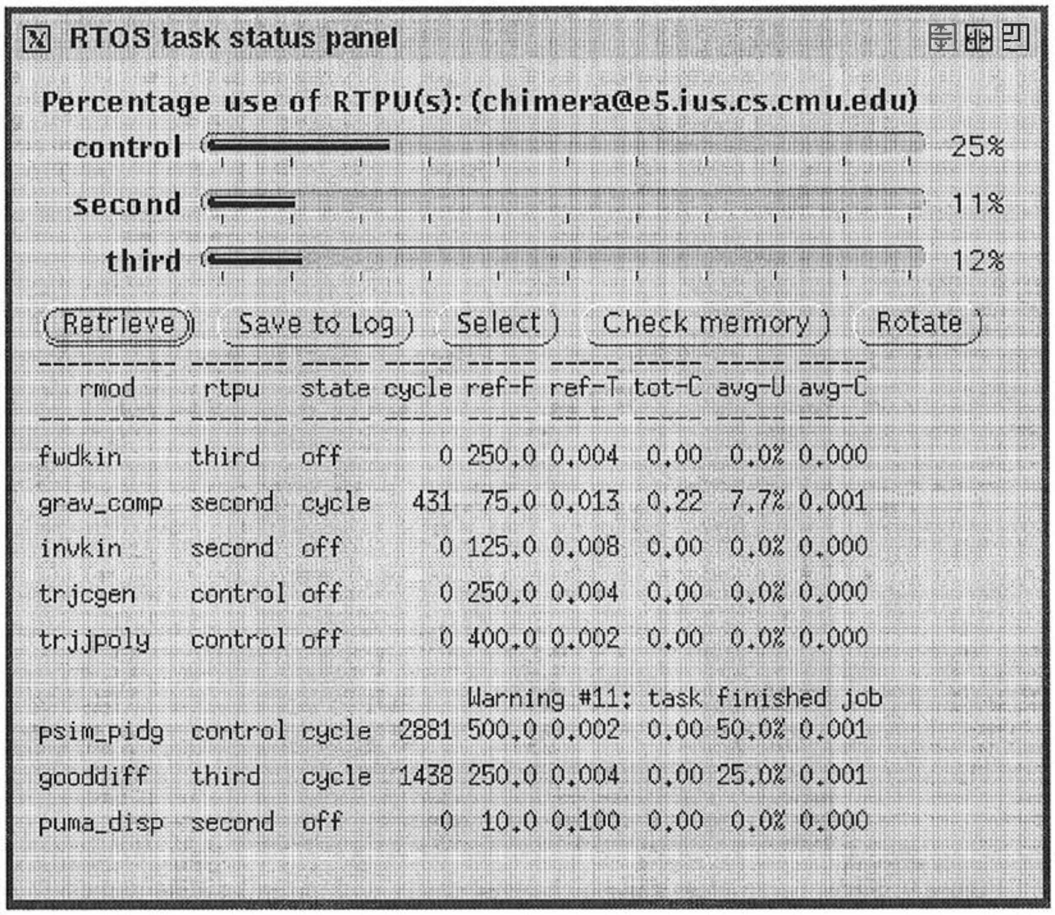
(Retrieve) (Save to Log) (Select)

```
Name        Value(s)
Q_MEZ  [6]:  -0.900,   -1.10,    2.90,    0.00,    1.30,   0.650
```

Figure 8: The Onika state variable display window.

window allows the user to view the performance statistics of the tasks, as well as the CPU usage (Figure 7). The user can also ask the RTOS to check the CPUs for memory corruption (an invaluable tool for debugging modules), save the current readings to a log file, or switch to viewing another subsystem. This information is not updated in real-time, but instead whenever something of importance occurs (i.e. a task is spawned, killed, cycled, deactivated, etc.) or whenever the user pushes the "Retrieve" button. Similarly, the state variable readout window displays the values of selected state variables at appropriate times or whenever the user specifies (Figure 8). Real-time display of variables is also supported within our reconfigurable software framework, but is implemented externally to Onika, to decrease the bandwidth which Onika must process. This is discussed further in Section 5.8.3.

The user may bring up a panel within which he or she may change the modifier values specified in the parameter file, both in the lexicon and on the canvas, as shown in Figure 5.



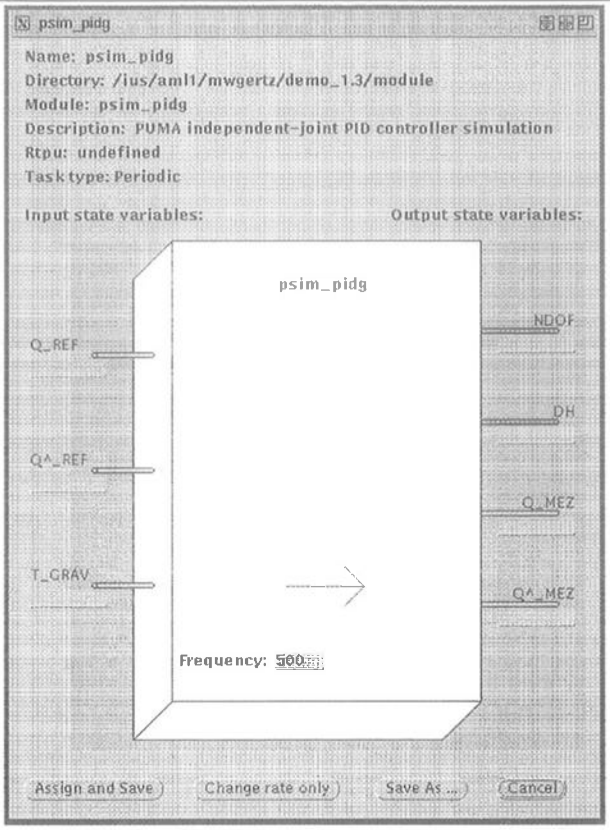
```
Percentage use of RTPU(s): (chimera@e5.ius.cs.cmu.edu)
  control                                          25%
  second                                           11%
   third                                           12%

---------- ------- ----- ----- ----- ----- ----- ----- -----
   rmod      rtpu  state cycle ref-F ref-T tot-C avg-U avg-C
---------- ------- ----- ----- ----- ----- ----- ----- -----
fwdkin     third   off       0 250.0 0.004  0.00  0.0% 0.000
grav_comp  second  cycle   431  75.0 0.013  0.22  7.7% 0.001
invkin     second  off       0 125.0 0.008  0.00  0.0% 0.000
trjcgen    control off       0 250.0 0.004  0.00  0.0% 0.000
trjjpoly   control off       0 400.0 0.002  0.00  0.0% 0.000

                              Warning #11: task finished job
psim_pidg  control cycle  2881 500.0 0.002  0.00 50.0% 0.001
gooddiff   third   cycle  1438 250.0 0.004  0.00 25.0% 0.001
puma_disp  second  off       0  10.0 0.100  0.00  0.0% 0.000
```

Figure 7: The Onika status window.

Besides being able to change the frequency of the task, the user is also able to change the names of the state variables associated with the tasks. This is especially useful when the programmer is using code developed at another site, where the naming conventions may not match his or her own laboratory's conventions. The user can enter the "desired" name of the variable beneath the "hard-coded" name of the variable. The modified task can then be used immediately without needing to recompile or reload the code.

4.2.2 Task combination rules

Within a task, any state variable can be declared as any of the following types: *in-const*, *out-const*, *in-var*, *out-var*, *in-both*, or *out-both*. Those of the *const* form are constants which are read or written at the initialization of a task, and never again accessed by that task. Those of the *var* form are read every task execution cycle, and so the values are assumed to change. Those of the *both* form read or write some initial value from the state


psim_pidg

Name: psim_pidg
Directory: /ius/aml1/mwgertz/demo_1.3/module
Module: psim_pidg
Description: PUMA independent-joint PID controller simulation
Rtpu: undefined
Task type: Periodic

Input state variables:
Output state variables:

Frequency: 500

Figure 9: Tasks parameters can be changed easily within Onika. The names in the I/O ports can be aliased without needing to recompile code.

variable table, but the values are assumed to change thereafter. It is possible that one task may declare a state variable to be constant, while another might declare it to be a variable. This might lead to certain problems. It would not make sense to have a task that expects, for example, a constant input to be connected to a variable output. To avoid such a possibility, a series of connection rules have been devised. These include: all types of inputs may connect with each other (that is, share the same state variable); no type of output may connect with another, to avoid race conditions; and inputs requiring initial values (*in-const*) may not connect to outputs which do not supply them (*out-var*).

Although a task might be considered connectable in the state variable sense, it still may be "unplaceable" due to conflict of modules or names. This is because the task names are used for task identification. Furthermore, running a module twice concurrently would be redundant and a waste of system resources. Tasks within the lexicon which cannot be legally placed on the canvas due to name or module conflicts are dimmed and made unselectable.

4.2.3 Creation of higher level routines

Before the combination of tasks can have be saved as a job, there must be exactly one output instance of each state variable used in the configuration. As mentioned in section Section 4.2.2, this is to ensure that each module can receive meaningful input.

When the user saves a configuration as a job for high-level users, Onika must determine whether or not the job routine to be created will require modifiers or not. In order to do this, Onika checks the configurations for tasks which require user input (typically a trigger task, such as the end location of a trajectory). If a task requiring user input is found, then any values it will need in the future as an upper-level job will be determined from the modifier icon which follows its icon. A job which requires a modifier is referred to as an *action requiring a target*, whereas a job which requires no modifiers is simply an *action*. The modifier of a job is referred to as a *target*.

Users can create (and modify) the images of jobs and targets using the icon definition window (shown in Figure 10). The visual cues indicating the syntax of the icon are added automatically by Onika. Once a job routine or target has been created, it is available for use in the upper level interface. The use of job routines and targets in the upper level is the subject of the next section.

4.3 Upper Level Interface

Similar to the lower level interface, the routines which may be used to create upper level applications are displayed to a user in one window, and assembled for later execution in another. Modifying icons (*targets*) are displayed in the same window as the available routines. This provides an easy mechanism for modifying any given routine. Jobs and targets are combined into a serial goal-oriented application at this level. The application can be saved at any time for later recall or modification. During execution, the task configurations associated with the jobs in the application are loaded into Onika and Chimera. The tasks are all spawned at the beginning of the application, and activated or deactivated as needed. As each job is completed, the system reconfigures into the next job.

Programmers at this level need not know anything about textual programming, controls, or how the controlled machinery operates.

4.3.1 Combining job routines

The basic unit of combination at the upper level interface is a *job*. A job is created at the lower level by combining tasks together (see Section 4.2.2). This functionality is hidden

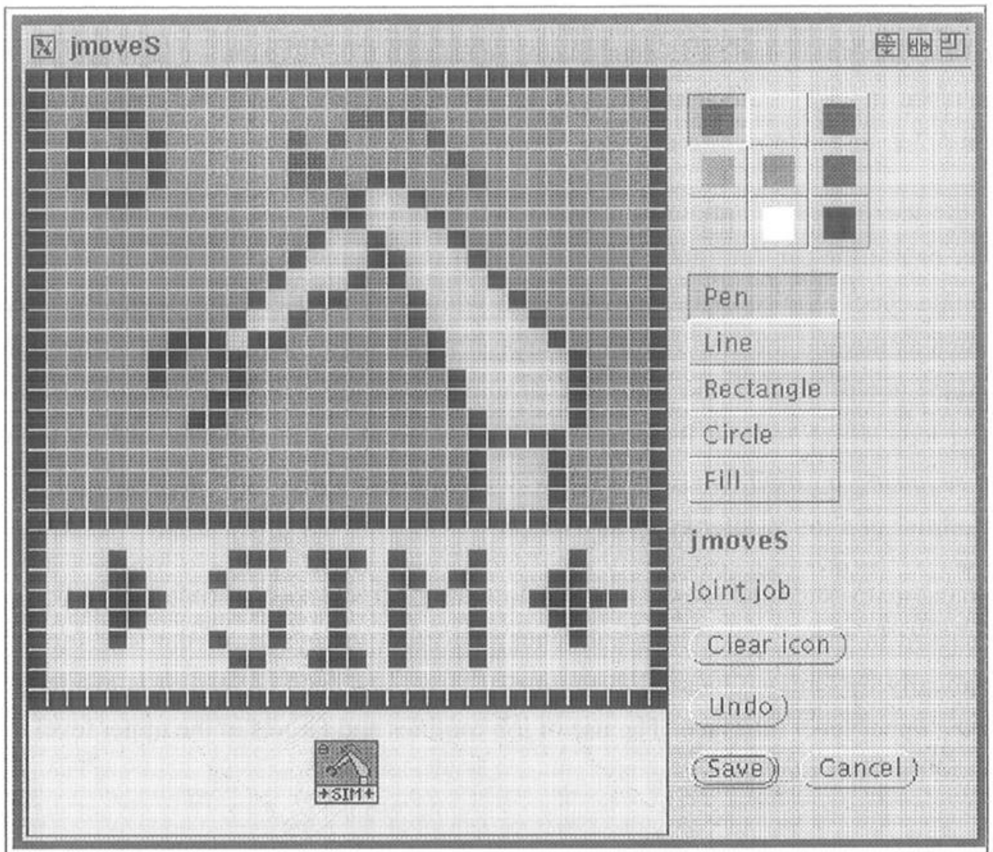
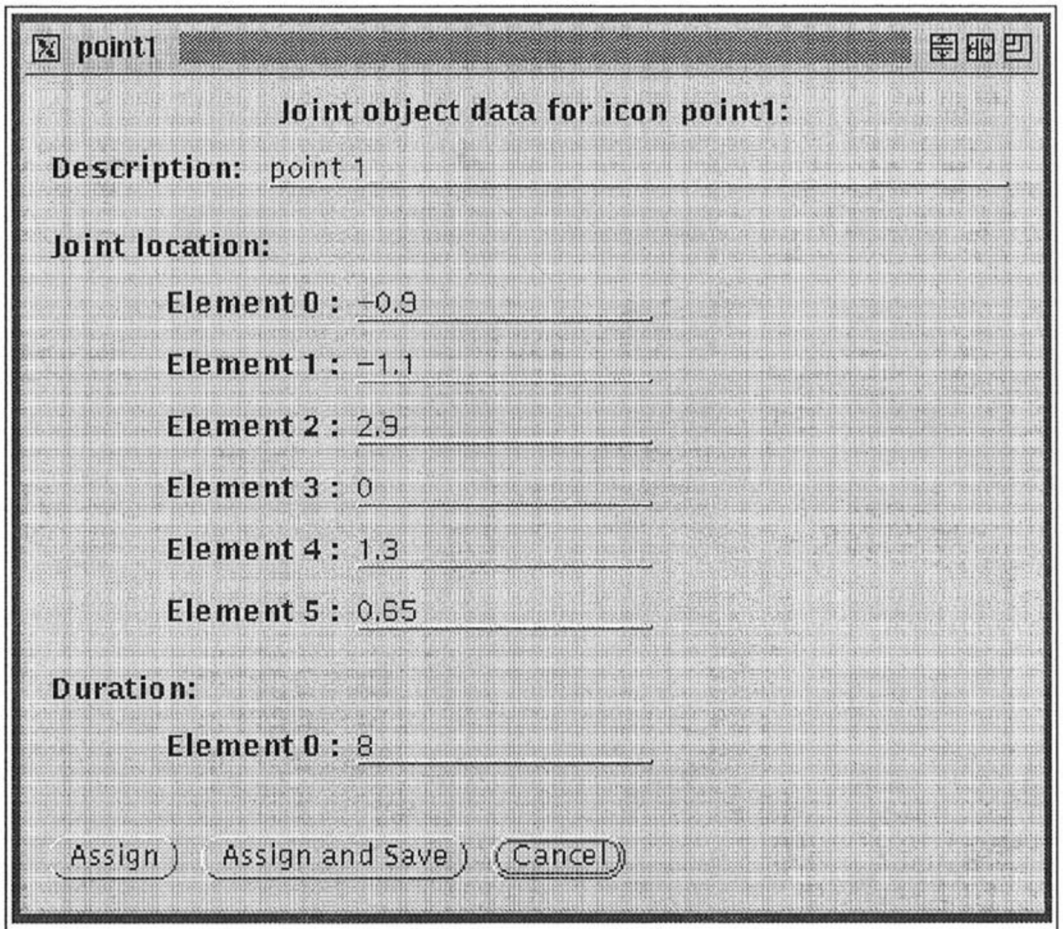


Figure 10: The user is saving the configuration of Figure 5 as a job, creating a picture for it. Onika has determined that this will be a "joint space job."

from the upper-level user, however. A job may or may not require a target, depending on how it was defined at the lower level. Jobs which require targets are referred to as *actions requiring a target*, whereas jobs which do not require targets are referred to simply as *actions*. An *action requiring a target* icon must be followed by exactly one target icon.

A target icon could be created for any state variable from the global state variable table. A preference file defines the types of targets which Onika will recognize. Targets can be created at both the lower and upper levels. The user supplies both the target type and its value(s) (Figure 11). These values are used by Onika when the job's trigger task is cycled, so that the user need not enter the input during the execution of the application (unlike the lower level, where information is entered by hand each time the task is cycled).

All icons are presented to the user in a *job dictionary*, shown in Figure 12. Each icon's picture is framed in a structure which has left and right edges of a certain shape and color. These edges are indicators as to which type of icon can sit next to another. Onika will not


point1

Joint object data for icon point1:

Description: point 1

Joint location:

Element 0 : -0.9

Element 1 : -1.1

Element 2 : 2.9

Element 3 : 0

Element 4 : 1.3

Element 5 : 0.65

Duration:

Element 0 : 8

Figure 11: Entering the information for a joint target.

allow non-interlocking icons to be placed next to each other. The icons may be drawn from the local file system, or automatically retrieved by Onika from remote filesystems. Multiple dictionaries may be open simultaneously during a session.

All targets have certain values associated with them, which can be changed by the programmer. These can be viewed and changed, both in the dictionary and in the application workspace.

4.3.2 Icon combination rules

Applications are assembled from the icons displayed in the job dictionary. This assembly is done within an *application workspace*. Icons are inserted from the dictionary into the application. If its edges match those of its potential neighbors, a new icon can be inserted between two icons. If the icon matches its left neighbor but not its right, a space is inserted between it and its right neighbor. The proper bridging icon can be inserted later into this

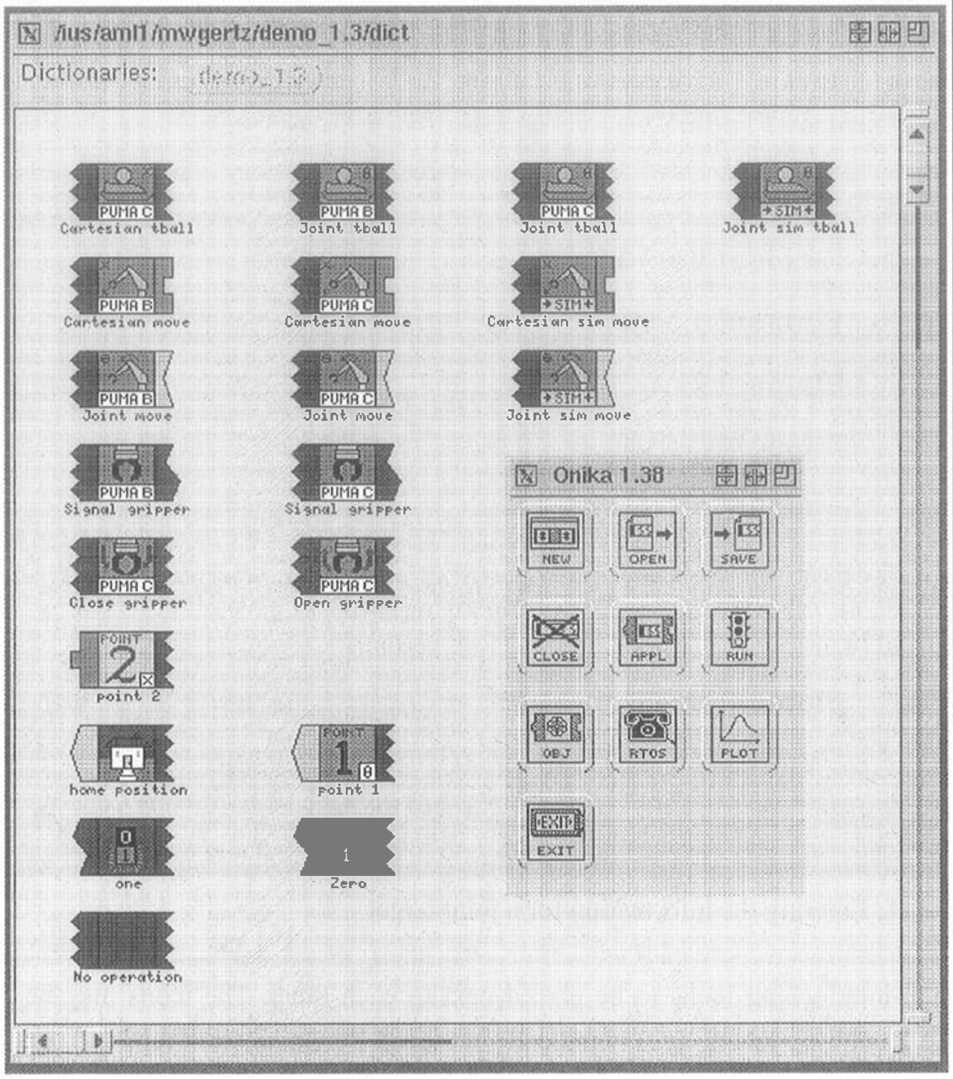
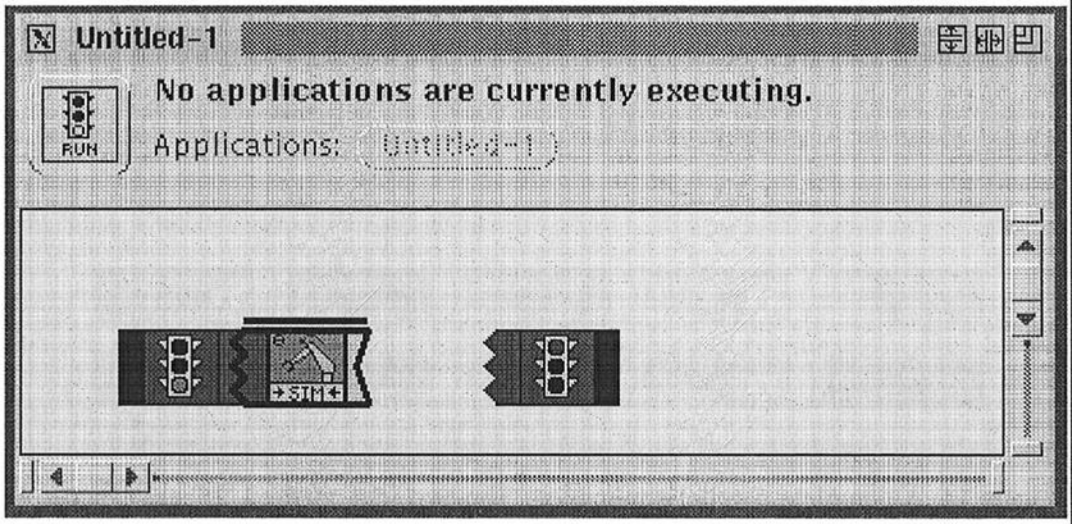


Figure 12: The Onika job dictionary (inset: the Onika control panel). Job and target icons are sorted according to type, and presented to the user with a description of their function.

gap (Figure 13). This process continues until the application is completed to the user's satisfaction. Icons may be inserted anywhere into an application, provided that they interlock properly with their potential left neighbor. Icons can also be deleted, or their information, descriptions, and images modified. An application will wrap-around at a certain length. Multiple applications can be opened at any time.

At run-time, applications can be completely controlled from within Onika, using a subsystem signaling mechanism discussed in Section 5.9. Onika will create any needed connections to the RTOS, and allows the user to pause applications, recover from errors, jump



No applications are currently executing.

Figure 13: An application in the process of being built. The “go light” and “stop light” icons indicate the beginning and ending of the application, respectively. The job created in Figure 10 has been inserted into the application; since its right edge doesn’t match the left edge of the icon which follows it, a hole was inserted. The hole must be filled by a joint target icon which starts out yellow and ends up red.

over jobs, assign breakpoints, or abort applications. Additionally, RTOS status and state variable feedback is displayed to the user.

Parallel applications are those applications which have two or more flows that can execute concurrently. The user declares an application to be a parallel application having *n* flows when creating a new application. Icons are placed within the various flows as normal. At run-time, Onika makes certain that enough connections to the RTOS exist to execute concurrent subsystems. Jobs within different applications can be synchronized so that they are guaranteed to begin execution together (Figure 14). This is a feature not found in any other real-time control system programming language, both visual or textual.

Applications created by combining jobs and targets can have icons assigned to them and can be used in other higher-level applications. Whereas “incomplete” applications (i.e. those with some target gaps unfilled) cannot be executed on a system, they can be iconified and used in other applications. “Incomplete” applications can be implemented as *actions requiring a target*, provided that any gaps within the incomplete application refer to the same type of *target* consistently.

Conditional branches are the least developed structural elements in Onika. They are implemented in the following way: when the user creates a new application, he or she

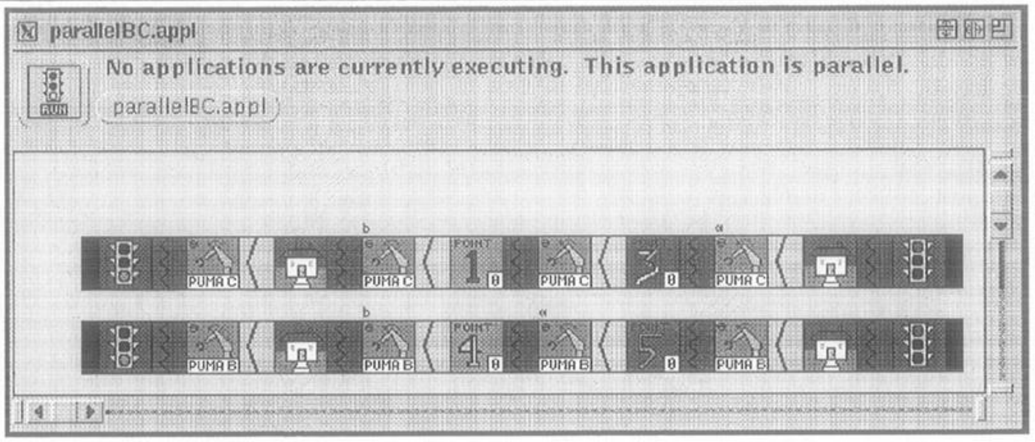

parallelBC.appl
No applications are currently executing. This application is parallel.
RUN
parallelBC.appl
b
a
PUMA C
POINT
1
3
PUMA B
4
5

Figure 14: A complete parallel application. The two jobs marked "b" are synchronized so that they will begin execution at the same time, as are those marked "a."

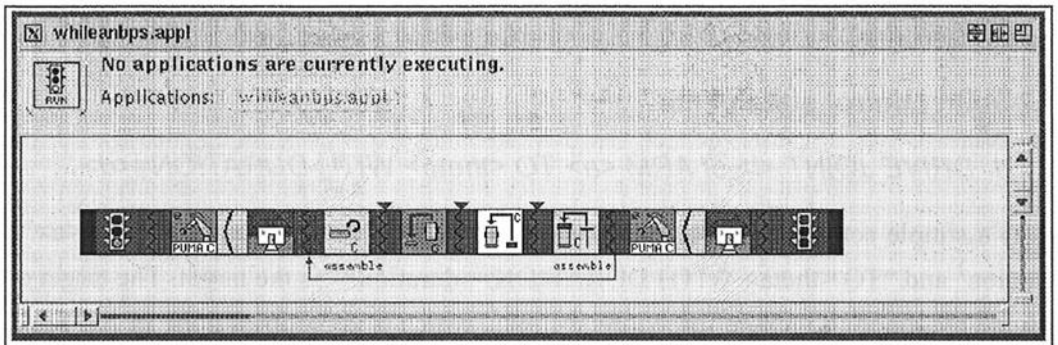
indicates that the application will be a conditional application having n branches. The user is then presented with n application flows in the application window, into which he or she can place appropriate icons for case 0, case 1, ..., case n-1. The entire application is then "iconified" and stored in the job dictionary. When used in another application, the flow taken in the nested conditional application depends on the return value of the job which executed before it. Future work on Onika will address a better means for presenting and implementing conditionals graphically.

Onika also includes support for both *top-test* and *bottom-test loops*. Two icons are "marked" as the starting and ending points for the loop, and exactly one of these icons is additionally marked as the test icon. When that icon's job completes itself, Onika analyzes its return signal to determine whether or not control should be passed to the other icon in the loop. Figure 15 shows an application with a loop and several breakpoints.

No-operation icons ("no-ops") are built into Onika as well. When encountered in an application, Onika treats them as complete jobs having a duration of zero. No-ops always return a "LOOP TRUE" signal if used as the test icon in a loop structure (allowing the easy creation of repetitive motion loops) and a "GOTO FLOW 0" signal if used before a conditional application icon. They are also very useful for spacing out icons to make a program more readable, and function like normal icons in all respects.

4.4 Completeness of the Upper Level Language

In this section, we compare the extended AL robot language developed for NASA's ROBSIM (Robot Simulator) program [46] to the syntax and grammar of Onika's upper level,


whileanbps.appl

No applications are currently executing.

Applications: whileanbps.appl

Figure 15: This application has a bottom-test loop tagged "assemble." It also has three breakpoints within the loop so that the user can step through the jobs included in the loop.

and show how all of AL's constructs would map into Onika's job-target visual language.[1] We have chosen AL as being representative of textual robotic languages in general. Its program commands are nearly identical to those of VAL II [63], for instance, except that the names of the AL commands are slightly more intuitive, and hence are used here. It should be noted that Onika's grammar is expandable beyond the commands listed here, whereas textual languages are, for all intents and purposes, static constructs. There are many other advantages to using Onika, including multiple subsystem control and real-time control.

4.4.1 Block statements

4.4.1.1 BEGIN-END

This is equivalent to the Start and Stop icons in Onika, which act as delimiters for applications, as well as for "case" blocks.

4.4.1.2 COBEGIN-COEND

This functionality is also covered by the Start and Stop icons. In AL, the first commands are guaranteed to start together, but no other synchronicity is possible beyond counting up job durations and adding appropriate pauses to certain arms while other arms finish up prerequisite tasks. With the synchronization markers available in Onika, we eliminate this guesswork, so that any two jobs in parallel flows can be guaranteed to execute simulta-

1. Certain extended AL constructs are not included in this comparison, since they have no parallels within our system. (for instance, arm initialization is never done from within a program, since it needs only to be done once after the robots are powered up). Numeric constructs are not compared, since all Onika numeric constructs are targets, the types of which are defined in a preference file and are not hard-coded.

neously.

4.4.2 Joint control statements

4.4.2.1 DRIVE JOINT <i> of ARM <a> TO <theta> WITH DURATION=<d>

This is a simple action/target combination, where "DRIVE JOINT <i> of ARM <a>" is the action[1] and "TO <theta> WITH DURATION=<duration>" is the target. The target can easily be redefined to include force/torque limits, which is sometimes done in AL as part of the DURATION vector. The joint command of Section 4.4.2.2 is preferable, since it is more general.

4.4.2.2 DRIVE JOINT <i> of ARM <a> USING <p> TO <theta> WITH DURATION=<d>

This is a variation on Section 4.4.2.1, where <p> is a polynomial to use during transit. At first, we included the "USING <p>" as part of the target, but truly modular code demands that this be part of the action instead (we cannot assume that the writer of the trajectory module has include cases for each type of polynomial). Of course, the user can redefine both the target and job to make <p> part of the target.

4.4.3 End effector control (Cartesian control)

4.4.3.1 MOVE ARM <a> TO <x> WITH DURATION=<d>

This is a simple action/target combination, with "MOVE ARM <a>" the action, and "TO <x> WITH DURATION=<d>" the target (with the same notes as to force/torque as Section 4.4.2.1). In AL, <x> is simply a position vector. We use normal, approach, and position vectors in our Cartesian targets, making our interpretation a little more general, and removing the need for Section 4.4.3.2.

4.4.3.2 MOVE ARM <a> TO <x> WITH YVECTOR=<y>, DURATION=<d>

This was a response on NASA's part to the command of Section 4.4.3.1 being unable to orient the wrist. Since we use normal, approach, and position vectors in our Cartesian targets, Section 4.4.3.1 is functionally equivalent to Section 4.4.3.2 with regard to our defined targets (other users may choose to define their target differently). In any event, "MOVE ARM <a>" is an action, and "TO <x> WITH YVECTOR=<y>, DURATION=<d>" is the target.

1. Note that the arm identifier is always included as part of the job in Onika, and is not a "parameter." An aliasing mechanism allows the user to select (with one mouse-click) the arm to be used if the default arm described in the job's configuration is inappropriate.

4.4.3.3 MOVE ARM <a> TO <x> USING SENSOR AT <loc> WITH VELOCITY=<v>, DURATION=<d>

A specialized NASA command, based on a simulated camera (the "sensor") mounted on the arm a distance of <loc> along the x-axis from the end-effector. We can postulate a more general case where <loc> is a Cartesian 6-element vector. "MOVE ARM <a> TO <x> USING SENSOR AT <loc>" is the action, "WITH VELOCITY=<v>, DURATION=<d>" is the target. The sensor may be part of the job's internal subsystem, or may be part of external subsystem; however, neither interpretation changes the action/target interpretation. This command is somewhat obsolete. In our system, we have defined a job-target combination that, if written in AL, would possibly be defined as "TRACK TARGET=<x> WITH ARM <a>," with "TRACK ... WITH ARM <a>" being the action, and "TARGET=<x>" being the target (perhaps a the name of a file having a description of the item to be tracked).

4.4.3.4 MOVE ARM <a> USING <xp> WITH DURATION=<d>

The Cartesian endpoint for the trajectory is defined by analyzing the six polynomials in <xp> (for *x,y,z,r,p,y*) and the duration; i.e. "Follow this path for <d> seconds." The action is "(PATH)MOVE ARM <a>" and the target is "USING <xp> WITH DURATION=<d>."

4.4.4 Base control

This extended AL command has been omitted from our library, since a base is simply treated as more degrees-of-freedom in our system, or is simply treated as another robot. NASA included these base commands so that they would not be limited by the 10-DOF constraint in their ROBSIM system. Onika (and Chimera, the real-time operating system which Onika uses) have no such limitations.

4.4.5 Hand control

4.4.5.1 CLOSE HAND <a>

This is a simple self-contained job requiring no target. In fact, the job has only one task, which closes a particular gripper.

4.4.5.2 OPEN HAND <a>

This is a simple self-contained job requiring no target. As with Section 4.4.5.1, the job has only one task, which opens a particular gripper.

4.4.6 Pause control

4.4.6.1 PAUSE ARM <a> WITH DURATION=<d>

This is another action/target tandem, with "PAUSE ARM <a>" the action and "WITH DURATION=<d>" the target. Note that are in fact many ways to pause in Onika. For instance, one arm may pause by reaching a synch tag, waiting for the other arm to reach its synch tag in its flow. Pausing can also be enforced by breakpoints, or by pressing a "pause" button during execution.

4.4.7 Flow Control Constructs

4.4.7.1 GOTO <label>

Not strictly implemented in Onika. The user can always skip the current step by pressing a button, and go to the next step. In a future version of Onika, we may add a "skip to" button to the execution control panel, which could be used during run-time. Actual *GOTO* functionality can be and should be achieved using conditionals. *GOTO* commands are cryptic, and their use even within textual programs is generally discouraged. We have avoided adding them to Onika so far, since their functionality is fully reproducible with the proper use of conditionals.

4.4.7.2 IF/THEN/ELSE

Onika fully supports this; in fact, in Onika conditionals are implemented as "case" statements, making them more general. When a conditional application icon is encountered within a higher-level application, the return value from the previous action is used to choose which "case" is followed at run-time. Future work on Onika will enhance the implementation and presentation of conditionals.

4.4.7.3 WHILE-DO

In Onika, the user marks two icons within an application flow with a unique tag, and chooses whether to make this a top-test or bottom-test loop. When the test icon is reached, its "finished" signal is analyzed for a "keep looping" or "continue" element, and execution flow is updated appropriately. Note that the trigger module in the end job must be programmed to return this signal when appropriate.

It should be noted that the ROBSIM extended AL has only top-test loops, not bottom-test loops, whereas Onika has both top- and bottom-test loops.

In the next section, we demonstrate how Onika has been used to research and develop the concept of a *distributed laboratory.*

4.5 Distributed Laboratories and Technology Transfer

4.5.1 Overview

The Onika visual programming environment has been used to research the concept of *distributed laboratories*[57][20], wherein both software and hardware resources can be shared across a network without needing to rewrite code[1]. Since Onika fully supports the Chimera methodology given in [16], differences in systems and code can be eliminated, leading to greater code compatibility between sites. However, a lack of hardware resources at a site can still lead to an inability to incorporate new ideas and technologies. To alleviate these problems, we propose the development of distributed laboratories to make both software and hardware readily accessible to researchers across networks such as the Internet or the World-Wide Web. Resources for use in distributed laboratories would be accessed by using *hypermedia* mechanisms.

We define the distributed laboratory as having the following qualities:

- *Distributed software libraries:* Reconfigurable real-time software modules are stored in object databases at various sites on the network. By using *hyperlinks*, retrieval of the software is transparent to the user regardless of its physical location.

- *Distributed hardware resources:* Various real-time control systems are accessible via the network on a time-share basis, so that a multitude of sites have access to equipment otherwise unavailable.

The advantages of a distributed laboratory include:

- *Increased technology transfer:* Software developed and debugged at one site can be stored in a software library, making it immediately available to other sites.

- *Zero logical distance:* The interface to a distributed laboratory need not be running on the same machine, nor even on the same filesystem, as their communications are all via hyperlinks across a network. The interface may be running on a machine across the room, or on another continent, without diminishing its ability to control a real-time system. Programmers inti-

1. Also referred to as *virtual laboratories* in previous publications.

mately familiar with their interface need not modify it to control any remote systems.

- *Expandability:* New hardware (remote or local) can be integrated into the system and controlled quickly. This can be done by replacing or adding a software module to an existing application, and leaving the rest of the modules in the application untouched.

- *Transparent simulation:* Using reconfigurable software modules, any real-time program can be simulated without changing any of the real-time code, as each hardware module has its simulated equivalent readily available in a software library.

- *Reduced costs:* If a participating site wishes to expand on the research of another site, but lacks the proper equipment, they can make use of the hardware resources available at a cost substantially lower than purchasing the equipment themselves. Furthermore, the site need not waste valuable time and money redeveloping code, since it would be available in one of the software libraries.

In the next two subsections, we show how Onika has been used to demonstrate the usefulness of distributed laboratories with respect to both hardware resources and software resources.

4.5.2 Shared Hardware Resources

Onika has been used several times to control a manipulator located hundreds of kilometers from the user, most recently in several demonstrations to top-level administrators and scientists at Sandia National Laboratories. A Sun 4 workstation running X11R5 with Internet access was made available to us at Sandia, on which Onika could be launched. The manipulator to be controlled (a PUMA 560 running in a Chimera environment) was located 2,600 kilometers away at Carnegie Mellon University.

Onika can either be run on the same file system as Chimera, or on a remote filesystem. In the latter case, Onika runs faster, but must upload any executables it compiles to the Chimera file system. In the former case, the executable is created on the same file system as Chimera, but the entire Onika display must be transmitted over the network to the host workstation (in this case located at Sandia). Both schemes have been used in remote demos in the past.

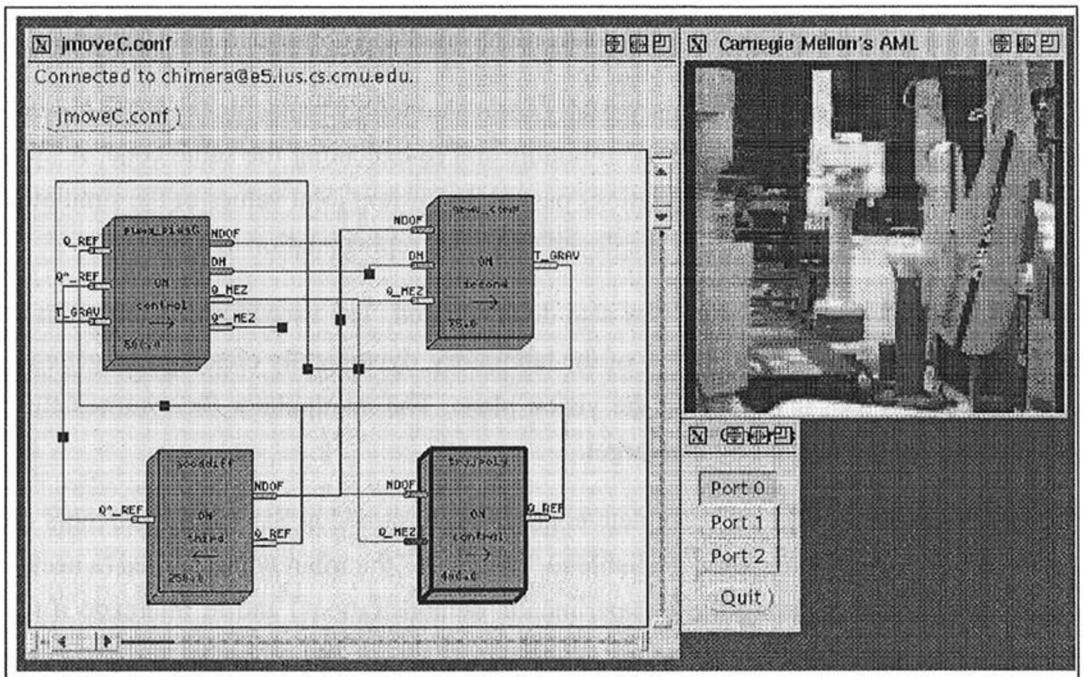


Figure 16: The distributed laboratory setup demonstrating control across a network.

Upon launching, Onika searched the user preferences and found two hyperlink anchors to libraries which the programmer used, one located "locally" at Carnegie Mellon, the other at Sandia. Using these hyperlinks, Onika downloaded both libraries from the network. For security reasons, the programmer was prompted for a password before being able to access the Sandia library. Once downloaded, Onika linked the required modules into a Chimera executable, which was stored in a Chimera-accessible location. Iconic hyperlinks to the modules were created and displayed in the library window.

The programmer then launched Chimera, and Onika connected to it with the click of a button. Using modules from both Sandia and Carnegie Mellon, the programmer created a joint motion job and activated all of its modules in less than a minute. Cameras located around the manipulator at Carnegie Mellon gave the programmer several different views of the manipulator, as 128x128 greyscale images were transmitted over the Internet at a rate of 10 Hz. The setup is shown in Figure 16.

Subsequently, the programmer used Onika to successfully demonstrate reconfiguration into (and the execution of) a pre-saved Cartesian motion job, and execution of an application which assembled a small DC motor. A demonstration of error recovery was also given, during which the "panic button" of the manipulator was pressed, interrupting a joint trajectory. Chimera successfully trapped the error, and notified Onika. Onika then automatically cleared the error, and the robot was reactivated and completed its trajectory.

Simulation of an application was successfully demonstrated by replacing the PUMA modules in an application with a simulation module and re-executing the same code. A synchronous module within the configurations passed the current joint values of the simulated robot to an external package which displayed graphical representation of the robot.

Semi-autonomous visual servoing was also demonstrated. The user clicked on a point in the window showing the camera view of the laboratory, dynamically creating a target containing the location of that point in the vision plane. The manipulator then immediately moved to the point specified by that target.

Throughout the demonstration, complete control was assumed at Sandia. Researchers on location at Carnegie Mellon were available to "power up" the robot when needed (a necessary safety precaution during these experimental demonstrations) and to intercede if the robot showed signs of instability in this experimental set-up, but otherwise did not interfere with the demo in any manner.

4.5.3 Shared Software Resources

The nature of the generic software modules in our laboratory's libraries is such that most of the code required to get new systems operating is already available to us. Using Onika and Chimera to assemble and control these modules, a previously-unused mobile robot (left over from a graduate student's project several years previous) was brought on line and visual servoing programs executed on it in less than two days. The only module which needed to be created for the mobile manipulator was the one that actually communicated with the robot's hardware; other modules, such as trajectory, kinematics, and visual servoing modules, were already available. A Utah-MIT hand located in our laboratory has also been brought on line in the same fashion, and no less than six other systems (including two Adept robots, two American robots, a Stewart platform, and the Reconfigurable Modular Manipulator System) either have been or will be brought on line over the next several months, bringing a total of eleven systems under shared software control in our laboratory.

The software libraries at Carnegie Mellon were recently used to launch new robotic systems at the Air Force Logistics Center (AFLC) in Texas. AFLC obtained the Chimera and Onika software and, using the software libraries, was able to get its systems up and running in less then two days. As enhancements to the Chimera real-time operating system and to the task modules have been made, AFLC has been able to download and immediately use these upgrades. Other laboratories are currently in the process of obtaining the

Chimera and Onika packages as well. As these sites do research using the reconfigurable software framework for reusable software modules, the task libraries available throughout the user community continues to grow.

4.6 Summary

The Onika visual programming environment presents interfaces suitable to both engineers and non-technically-oriented persons, without a loss of scope in the language. In the lower-level interface, programmers combine icons representing reconfigurable generic tasks into job configurations. These job configurations can be fully controlled from within Onika, and feedback from the supporting RTOS is displayed to the user. A background syntax checker prevents certain impossible configurations from being created. The jobs created at this lower-level can be represented as high-level icons, with Onika automatically determined the syntax of the newly-created icon. These job icons can be combined with target icons to create goal-oriented applications in the upper level. Both the syntax and grammar of the upper level language are expandable, and the functionality of general robotic textual languages can be completely supported within Onika. The syntax of the upper level icons is made clear from visual cues, and a background syntax checker eliminates the possibility of creating syntactically-incorrect applications. Applications can be executed and completely controlled within Onika, and the user is given appropriate feedback during execution. Applications can be "iconified" and used in higher-level applications. Conditionals, parallel branches, top- and bottom-test loops, and breakpoints are all supported within Onika. Additionally, jobs in parallel-executing applications flows can be synchronized, so that explicit pause commands are not needed to synchronize a parallel application. Onika supports both shared hardware and software resources within the context of a *distributed laboratory*. The software libraries developed by our laboratory are currently in use at various other sites around North America. Onika fully supports, and is fully supported by, the Chimera 3.2 Real-Time Operating System.

5. Implementation of Onika

5.1 Introduction

In this chapter, we discuss the specifics of the mechanisms which implement the functionality of Onika 1.41, as well as pointing out possible directions for future work. For real-time control of tasks, Onika fully supports, and is fully supported by, the Chimera 3.2 Real-Time Operating System [59]. Both systems were developed concurrently under the same reconfigurable software architecture, ensuring proper interaction between the RTOS and the interface [57].

In Section 5.2, we describe how control tasks are retrieved and represented within Onika, as well as how allowances are made for different naming conventions between different laboratories. In Section 5.3, we describe how configurations can be built and cycled on the RTOS. The upper level dictionary is presented in Section 5.4, and the creation of applications is discussed in Section 5.5. Icon creation is discussed in Section 5.6, whereas user preference handling for both the upper and lower level is presented in Section 5.7. Section 5.8 introduces the various communication mechanisms developed for Onika. The execution loop is discussed in Section 5.9, in which we also discuss how the simulation of applications is achieved in Onika, while error handling is discussed in Section 5.10. We discuss the portability of the Onika algorithms to other systems in Section 5.11. Finally, in Section 5.12, we summarize this chapter. We regret that the source code for Onika cannot be included in the appendices to this dissertation, as is normally done for code developed during the course of research, since Onika exceeds 25,000 lines of code, and therefore would require nearly a thousand pages to print out.

5.2 Task Representation

In this section, we discuss how tasks represented, implemented, and used within Onika.

We also present our "aliasing" mechanism, which allows code generated at sites with different naming conventions to be used immediately without needing to recompile or relink code.[1]

5.2.1 Loading tasks

Onika may be launched in such a way as to automatically search for a Chimera connection, or the user may choose to run an Onika session without any Chimera connections. In the prior case, Onika asks Chimera for the locations of desired task software libraries; in the latter case, Onika gets this information from the UNIX environmental variable *$ONIKA_LOCAL*. These locations may refer to directories on the local file system, or may refer to directories on remote file systems on the network (as discussed in Section 5.8.4).

Given the locations of these libraries, Onika searches them for files with the suffix *.rmod*. These files contain all of the information necessary to create an icon "on-the-fly" for a given task. An example of such a file is given in Code 1 on page 27. Starting with the first *.rmod* file in the first library, Onika loads the information in this file into a *TASK* node (shown in Code 2). The size of the icon is then determined by assigning it a base size and iteratively increasing this size by steps until the distances between the I/O pins on either side of the task's icon are tolerably far apart when the icon is drawn. (The base size, step size, and minimum pin separation are all user-defined, as discussed in Section 5.7, whereas the width of any icon is the same for each icon and is hard-coded into Onika.) This height information is also stored in the *TASK* node. The node is added to a double-linked list of *TASK* nodes for a given library, and assigned an (x,y) location to the right of the previous icon (wrapping around as required). Each software library opened has its own linked list of nodes. The library information, in turn, is stored in a node and entered into a double-linked list of library nodes. (Library nodes are the same as *CONFIG* nodes, described in Section 5.3.) Thus, node traversal from a task in one library to a task in another library is fast and simple.

Task icons must have unique names. If the name of a task matches that of another task already loaded into any library-list, the task will not be added to the current library-list.

When all tasks have been loaded, Onika refreshes the library window, thereby causing the icons to be drawn. Only one library can be seen in the window at a time; this is the library pointed to by the global variable *CurrentLexicon*.[2] The user can view different libraries at

1. The Chimera implementation of this "aliasing mechanism" was discussed in [57]; the two were developed simultaneously and cooperatively.

will by selecting the appropriate library button at the top of the window; this has the effect of changing the value of *CurrentLexicon*. To present a library visually, Onika traverses the library's linked list of task nodes, starting at the first node. Given the height and width, Onika draws the icon's rectangle and adds 3D detail. Next, it draws the I/O pins with the proper colors. The color of the I/O pins is dependent on the type of state variable they represent. *INCONST* and *OUTCONST* state variables (which have some initial value, but are not periodically variable) are orange, *INVAR* and *OUTVAR* state variables (which have no meaningful initial value but are updated periodically) are colored yellow, and those state variables listed as both *INVAR* and *INCONST* or both *OUTCONST* or *OUTVAR* for the task are considered by Onika to be *INBOTH* or *OUTBOTH* respectively, and are assigned only one pin (instead of two, which would graphically confusing) which is colored blue. Finally, Onika draws the name of the task in the center of the icon, an arrow indicating of direction of data flow (input to output) beneath the name, and its frequency or period in the left lower corner of the icon. On a Sparc 10, the refresh rate is roughly 0.025 sec/icon, making even large libraries renderable in an acceptable amount of time. The refresh routine also redraws the icons in the currently displayed library every time an icon is selected or deselected.

An "infinite" event loop assigned to the library window catches all mouse events generated by the user. These events are evaluated, and those found meaningful are processed appropriately (for instance, clicking an icon with the left mouse button "selects" that icon). Appendix B.1 describes which mouse events have meaning within the library window.

A pop-up menu is provided at the top of the library window so that the user can select tasks by name if he or she does not care to search the icons visually for a particular task. The menu includes all tasks in all libraries, and is generated automatically using node traversal after all libraries have been loaded. When a task is selected from the menu, the library window is automatically scrolled to display that task's icon, which will be rendered as having been "selected." If the task is not in the currently displayed library, the library is identified by a traversal search routine, the value of *CurrentLexicon* is updated, the library window is scrolled to the appropriate task and refreshed with the task rendered as "selected."

2. The variable is called *CurrentLexicon* for historical reasons, since "libraries" were originally called "lexicons" in Onika.

```
typedef struct tnode {
    struct tnode *prev,*next;     /* For use in linked lists */
    NAME_STRING name;             /* Name of the task (no .rmod) */
    DIR_STRING dir;               /* Directory of the task */
    NAME_STRING module;           /* The actual object file (no .o) */
    char desc[MAXLINELEN];        /* The description -- I don't use */
                                  /*   it, but I need to save it */
    int lx,ly;                    /* Preferred icon location */
    int show;                     /* Can it be selected */
    int inleft;                   /* Are inputs on the left */
    int jobtype;                  /* Will it need UI? */
    int spawned;                  /* Is the task spawned? */
    int stask;                    /* What is its RTOS task number? */
    NAME_STRING rtpu;             /* What rtpu is it on? */

    float freq;                   /* The frequency of the task */
    int  type,                    /* Periodic or aperiodic */
         active;                  /* Is the node on or off on RTOS? */

    int size;                     /* Height of the icon */
    int num_inconsts,             /* Number of INCONSTs & INBOTHs */
        num_invars,               /* Number of INVARs */
        num_outconsts,            /* Number of OUTCONSTs & OUTBOTHs */
        num_outvars;              /* Number of OUTVARs */
    int input_types[MAX_IN_L3];   /* What svar types are the inputs?*/
    NAME_STRING
        input_svars[MAX_IN_L3];   /* What are their names? */
    int output_types[MAX_OUT_L3];
                                  /* What svar types are the outputs? */
    NAME_STRING
        output_svars[MAX_OUT_L3];/* What are their names? */
    int num_svaraliases;          /* Number of SVARALIASES */
    NAME_STRING coded[MAX_IN_L3+MAX_OUT_L3];
                                  /* Name in C-code */
    NAME_STRING svaralias[MAX_IN_L3+MAX_OUT_L3];
                                  /* Name to use instead */

    } TASK;
```

Code 2: The structure of an Onika *TASK* node. The field *show* is only used when the task is in a library; the fields *inleft*, *spawned*, *stask*, and *rtpu* are used only when the task is in a configuration.

5.2.2 Manipulating tasks

Tasks can be manipulated in many ways. They can be selected, dragged, and modified, and their files can be viewed and edited.

To select a task icon, the user moves the mouse cursor over the icon, and presses the left mouse button. The user then releases the mouse button or, if the icon is to be dragged, moves the mouse cursor to some other location in the window and then releases the mouse button. If an icon is dragged, a rectangular outline follows the cursor around until the mouse button is released to indicate to the user where the icon will be repositioned. The window is refreshed after an icon is selected or dragged.

To modify an icon, the user moves the mouse cursor over the icon, and presses and releases the middle mouse button. A window will appear with an enlarged picture of the icon, allowing the user to change the frequency of an icon or change its SVAR aliases (discussed in Section 5.2.3). The user can save these changes to the *.rmod* file (necessary if SVAR alias changes are made, since the underlying RTOS requires any SVAR changes to be saved to the file), or accept changes without saving them (if only the frequency was changed).

By using the *Shift* key or *Control* key with the middle mouse button, the user can view the module source code or *.rmod* file respectively. This is done by issuing an *xterm* system command which runs the program *$ONIKA_VIEWER* (which, by default, is the program *more* but could be, for instance, *emacs*) with the appropriate file name as an argument. If the module source code is unavailable, attempts to view/edit it are ignored.

5.2.3 Aliasing to local conventions

As mentioned in Section 5.2.1 and Section 5.8.4, Onika can retrieve and integrate (at launch time) tasks created and made available at other sites on the Internet. However, different sites often have different naming conventions. For instance, one site may call the global state variable which stores the degrees-of-freedom of a robot *NDOF*, whereas another site may simply call this variable *N*. This is a problem, because the task in question will not properly connect, either graphically or functionally, with other modules using the degrees-of-freedom of the robot as an input or output. As another example, the user may wish to have a generic differentiator, and use it with a wide variety of state variables, rather than having to build a differentiator task for each type of state variable. Normally, because the names of state variables are hard-coded into a task's module object code, the user would be forced to obtain the source code of the software, make the necessary naming changes within it, recompile the code, and relink the code into the RTOS executable, all of which is tedious and error-prone.

To address this problem, Onika supports state variable (SVAR) aliasing,[1] wherein the names of the input and output SVAR ports of a task can be assigned a new name without needing to recompile or relink any code. By clicking on a task's icon with the middle mouse button, the user can bring up a window containing an enlarged view of the icon, with the hard-coded names and any "assigned" names of each state variable shown at each of the corresponding "pins" on the icon (shown in Figure 5 on page 29). The user can modify the assigned names as needed and "save" any changes made to the task's *.rmod* file. The changes are reflected immediately within the library window, and Onika informs the RTOS that the SVAR hitherto referred to as a certain name in this task will now be referred to by a new name. The task is thus usable immediately, without needing to alter the object code of the task at all.

The arrays *coded* and *svaralias* in the *TASK* node contain the hard-coded names and "new" names of the SVARs of task, respectively. When the tasks are initially loaded in, the list of operational aliases for the task (if any) are loaded into these fields first, in the order in which they appear in the file. Later in the task's loading procedure, when the SVAR names for the inputs and outputs are loaded, the names are checked against the *coded* array to determine if they are to be referred to by an alias. If so, the alias is written to either the array *input_svars* or *output_svars* (as appropriate); if not, then the coded name is written instead. The arrays *input_svars* and *output_svars* are then used when determining connectivity and when drawing the names of the SVARs above the icon's pins. This eliminates the need to check within the icon rendering subroutine as to whether the SVAR name should be drawn as "coded" or "aliased," speeding up the rendering process.

When the icon is "expanded" to modify the aliases of the input and output pins of an icon, the names in *input_svars* and *output_svars* are checked individually to see if they are aliases or the actual "coded" names using a list search. If a name is the actual "coded" name, then it is presented as such, and the space for the variable's alias is left blank. If not, then the name is written in the alias's space, and the alias's hard-coded counterpart is presented as the coded name. The user can modify the alias's space for each SVAR. When saved, the arrays *coded* and *svaralias* are updated as necessary, and the values of the

1. Not to be confused with Onika's built-in "task aliasing mechanism," which allows high-level applications to be run in simulation or on different real-time control systems without changing an application. This mechanism is discussed in Section 5.9.1.

arrays *input_svars* and *output_svars* are updated from the alias's space for the appropriate pin if non-blank, or from the "coded" name if no alias is specified. Using this mechanism, it is perfectly possible (and potentially useful) to create a situation where the name *A* is aliased to *B*, and *B* is aliased to *A,* without fear of looping infinitely during alias analysis, as the aliases are resolved only once at any time.

Modifications to aliases can be performed on tasks in both the library and the configuration workspace (discussed in the following section).

5.3 Configurations

The tasks loaded into the task libraries can be modified and manipulated to an extent, but in order to actually control a task, it must be spawned within the configuration workspace. To do this, the user selects a task within the library using the left mouse button, moves the mouse cursor to a desired spot within the configuration window, and presses the right mouse button. The icon is rendered at the desired spot, connections to other icons are automatically made, and the user is prompted for "spawning" information as needed. In this section, we present the algorithms which govern the building and control of configurations.

5.3.1 Configuration formats

The format for configuration nodes (*CONFIG* nodes) is given in Code 2. The *CONFIG* node is also used for library nodes, but in the latter case many of the fields are unused. The configuration nodes are contained in a double-linked list, with only one configuration in view at any given time (pointed to by *CurrentConfig*). As with the library window, the user can select the desired configuration for viewing from a row of buttons at the top of the window, changing the value of *CurrentConfig* and causing the configuration window to be refreshed. At least one configuration must be open at any given time. The user may create new configurations, or can load in pre-saved configurations (the file format for configurations is shown in Appendix A.2).

5.3.2 The starting configuration

At the beginning of the session, the user starts out with a blank configuration labeled "Untitled-1." If a connection with the real-time operating system is attempted when launching Onika, then the RTOS will be queried as to its current state, and the blank configuration will be updated to show any tasks which might be already spawned on the system. If the RTOS connection is made after tasks have already been placed into the configuration window, Onika will juxtapose the current configuration in the configuration

```
typedef struct config {
    struct config *prev,*next;    /* For use in linked lists */
         CANV_STRING name;        /* Name of the configuration */
         DIR_STRING dir;          /* Directory of the configuration */
    TASK *task_header;            /* Header of task linked list */
    TASK *task_tailer;            /* Tailer of task linked list */
    TASK *selected;               /* Selected icon within config */
    LOCUS *locus_header;          /* Header of locus linked list */
    LOCUS *locus_tailer;          /* Tailer of locus linked list */
    Panel_item button;            /* Button used to select config */
    int dirty;                    /* Have task changes been made? */
    int cdirty;                   /* Have cosmetic changes been made? */
    int elements;                 /* Number of tasks in config */
    char sbsname[128];            /* Its RTOS subsystem name */
    ENET *socket;                 /* Its RTOS socket */
    ENET *sigsocket;              /* Its signal socket */
    int sockmode;                 /* Active or inactive sockets? */
    char *rtpu_list;              /* List of available RTPUs */
    Frame rtpu_frame;             /* How they are displayed */
    Panel_item choice_list;
    } CONFIG;
```

Code 3: The structure of an Onika *CONFIG* node. When used as a library node, only the fields *prev*, *next*, *name*, *dir*, *task_header*, *task_tailer*, *selected*, and *button* are used; the rest are useful only for configurations.

window with those tasks already cycling on the RTOS, so that the resultant configurations is given by the equation:

$$T_f = (T_o \cup T_r) - S(T_o, T_f) \qquad (1)$$

where T_f is the final set of tasks which will be in both the configuration window and the RTOS subsystem, T_o is the set of tasks initially placed within the configuration window, T_r is the set of tasks initially spawned on the RTOS subsystem, and $S(T_o,T_f)$ is a subset of tasks within T_o which conflict with tasks in T_r, as discussed in Section 5.3.3. To integrate to two sets of tasks, Onika first requests the set T_f from the RTOS, and uses this to determine $S(T_o,T_f)$. The tasks $T_o \cap S(T_o, T_f)$ are removed from the configuration's linked list of *TASK* nodes[1], and the tasks $T_f - T_o$ are added to the window, with their RTOS data stored in their respective new *TASK* nodes. Next, the tasks $T_o - T_f - S(T_o,T_f)$ are spawned on the RTOS, with the user being prompted for target RTPUs as required. The remaining tasks, $T_o \cap T_f$, are already present in both the configuration window and on the RTOS subsystem, so that their *TASK* nodes on the Onika side need only be updated with their RTOS data. Finally, the screen is refreshed, and the icons are redrawn.

5.3.3 Laws governing the placement of tasks

Tasks may not simply be placed into a particular configuration without regard to the tasks already present, as certain tasks may interfere with the operations of other tasks. In particular, tasks which share output state variables may not co-exist within the same configuration, since a race condition would result in which the value of the shared state variable would no longer be predictable. (Sharing input state variables is inconsequential, since these are "read-only".) To avoid this possibility, tasks which cannot be placed in the current configuration are "greyed out" in the current library (or drawn with dotted lines if a monochrome monitor is used). This is done by setting the value of the field *show* in the appropriate TASK node to zero, and refreshing the library window. The tasks which are "greyed out" comprise a set given by the equation:

$$G(T_o, T_l) = \{\forall t \in T_l | (\exists s \in T_o)(((O(t) \cap O(s) \neq \emptyset) \vee (s = t) \vee M(s, t))\} \quad (2)$$

where:

$$M(s, t) = (\forall t \in T_l | (\exists s \in T_o)(((O_V(t) \cap I_C(s) \neq \emptyset) \cap (O_V(t) \cap I_C(s) \neq \emptyset))) \quad (3)$$

and where T_o is the set of tasks in the current configuration, T_l is the set of tasks in the library, t and s are tasks in T_l and T_o respectively, the function $O_V(x)$ returns the set of *OUTVARs* of a task x not part of an *OUTBOTH* tandem, $I_C(x)$ returns the set of *INCONSTs* for x not part of an *INBOTH* tandem, and the function $O(x)$ returns the set of all output SVARs associated with x (any aliases for the outputs are used in place of the "hard-coded" names where applicable whenever this equation is considered). The set of greyed-out tasks for each library is updated whenever the current configuration is changed so that a different configuration is viewed, whenever a task is added to the current configuration, or whenever reconfiguration is performed (Section 5.3.4).

Two tasks are considered equivalent if they have the same name (and thus the same sets of I/O SVAR pins, since Onika will only load in one task for any given name during the library build at the beginning of the session). Note that, by default, if $s=t$ in Equation (2), then t would seem to be an element of $G(T_o,T_l)$ by the $(O(t) \cap O(s) \neq \emptyset)$ clause, and

1. We could just as easily remove the tasks from the RTOS subsystem which conflict with the tasks in the Onika configuration window, giving preference to the Onika tasks over the RTOS tasks, instead of the other way around. However, the presumption is that the tasks already on the RTOS may be involved in critical work which should not be interfered with, and thus preference is given to them rather than the hitherto non-operative tasks placed into the Onika configuration window.

thus having the (s=t) clause would at first appear to be redundant. However, since it is perfectly possible that a task might have no SVAR outputs (for instance, a task which simply logs input data to a file), there may arise instances where the "intersection of outputs" clause is insufficient to keep tasks of the same name from being duplicated within a configuration. Since both Onika and its supporting RTOS Chimera relying on task names to identify and control processes, this is a problem. We therefore supplement Equation (2) by including the (s=t) clause to catch those rare instances where a task has no outputs.

Equation (3) is true when, if the a library task has one or more state variables declared as variable outputs (*OUTVARs*), the exists a task in the current configuration which has those same state variables declared as constant inputs (*INCONSTs*), or vice-versa. This is considered an illegal connection within our software framework.

For the merging of configurations described in Section 5.3.2, we need to determine whether conflicts exist between tasks on the Onika side and tasks on the Chimera side, and remove tasks on the Onika side which do conflict (given in Equation (1) as $S(T_o, T_f)$). However, it would in this case be wrong to destroy identical tasks present on either side, since the point is to have a graphical instance on the Onika side and a spawned instance on the RTOS side for each task. If $G(T_o,T_f) = S(T_o,T_f)$ and task t existed in both Onika and Chimera at start-up, then the graphical instance of the task in Onika would be destroyed, and a new one would have to be created. We could modify Equation (1) to do this, but it is simpler (and less expensive computationally) to simply define $S(T_o,T_f)$ to be:

$$S(T_o, T_f) = \{\forall t \in T_l | (\exists s \in T_o)(((O(t) \cap O(s) \neq \varnothing) \vee M(s,t)) \wedge (s \neq t))\} \quad (4)$$

The Onika user also has the option to have the interface give warnings whenever questionable situations arise; for instance, if a task is loaded in which has an SVAR listed as both an output and an input, if a configuration is being saved as a high-level job while still having hanging inputs, and so on. These "strange" situations are not prevented, as feedback from our engineers indicates that sometimes "normal" conditions must be violated during testing to determine the safety of modules. The following situations are flagged if the environmental variable *$ONIKA_WARNINGS* is set:

$$W = ((\exists t \in T_l)(O(t) \cap I(t) \neq \varnothing)) \quad (5)$$

$$W = (\exists p \in \Sigma | ((p \in I(\exists t \in T_l)) \wedge (p \notin O(\exists s \in T_l)))) \quad (6)$$

where the value of W is *TRUE* when a warning should be given, p is a state variable, and Σ is the set of all state variables in the system.

Equation (5) if true when a task has one or more inputs which are also declared as outputs in the same task. This equation is evaluated when tasks are loaded into Onika at the beginning of a session, so that the user can be alerted to tasks which generate their own inputs.[1]

Equation (6) is true when, given the set of tasks that constitute a configuration, there exists at least one input SVAR in at least one task which does not have a corresponding output SVAR in any of the tasks in the configuration. This is known as the "hanging input" condition. This equation is calculated whenever the user attempts to save the configuration as a higher-level job (discussed in Section 5.6.1).

5.3.4 Reconfiguration

Reconfiguration describes the process wherein one configuration is changed for another. There are two types of reconfiguration: *static reconfiguration*, wherein the user builds a configuration by selecting modules for placement into a blank configuration, and *dynamic reconfiguration*, wherein modules in an existent configuration are deactivated or deleted when not needed in the next configuration, and modules required in the new configuration are spawned or activated as needed [60].

Within Onika, static reconfiguration is performed by selecting tasks in the library and placing them into the configuration window; assuming that a RTOS session is in progress, the task is spawned on the underlying RTOS in an inactive (i.e. non-cycling) state. Static reconfiguration is performed manually by the user, or automatically by Onika when previously-saved configurations are opened, from within Onika's lower level interface.

Dynamic reconfiguration is performed in Onika in two ways: *subset dynamic reconfiguration*, and *superset dynamic reconfiguration*. Both types can be performed manually by the user, or automatically by Onika, although manual subset dynamic configuration is limited to the lower level interface of Onika.

Subset dynamic reconfiguration is pictured in Figure 17.It is used primarily when reconfiguration is no knowledge of future configurations is available; for instance, when an engineer is interactively designing configurations within Onika's lower level. Given a starting configuration T_1 and a desired configuration T_2, Onika performs automatic subset dynamic configuration as follows: first, the configuration T_2 is loaded from a pre-saved file, and set of tasks T_2 - T_1 are spawned on the RTOS. Next, the set of tasks T_1 - T_2 are

1. There is no logical reason to create such a task, since task modules are capable of storing previous values internally, thereby eliminating the overhead required to access the global state variable table for the appropriate value every cycle.

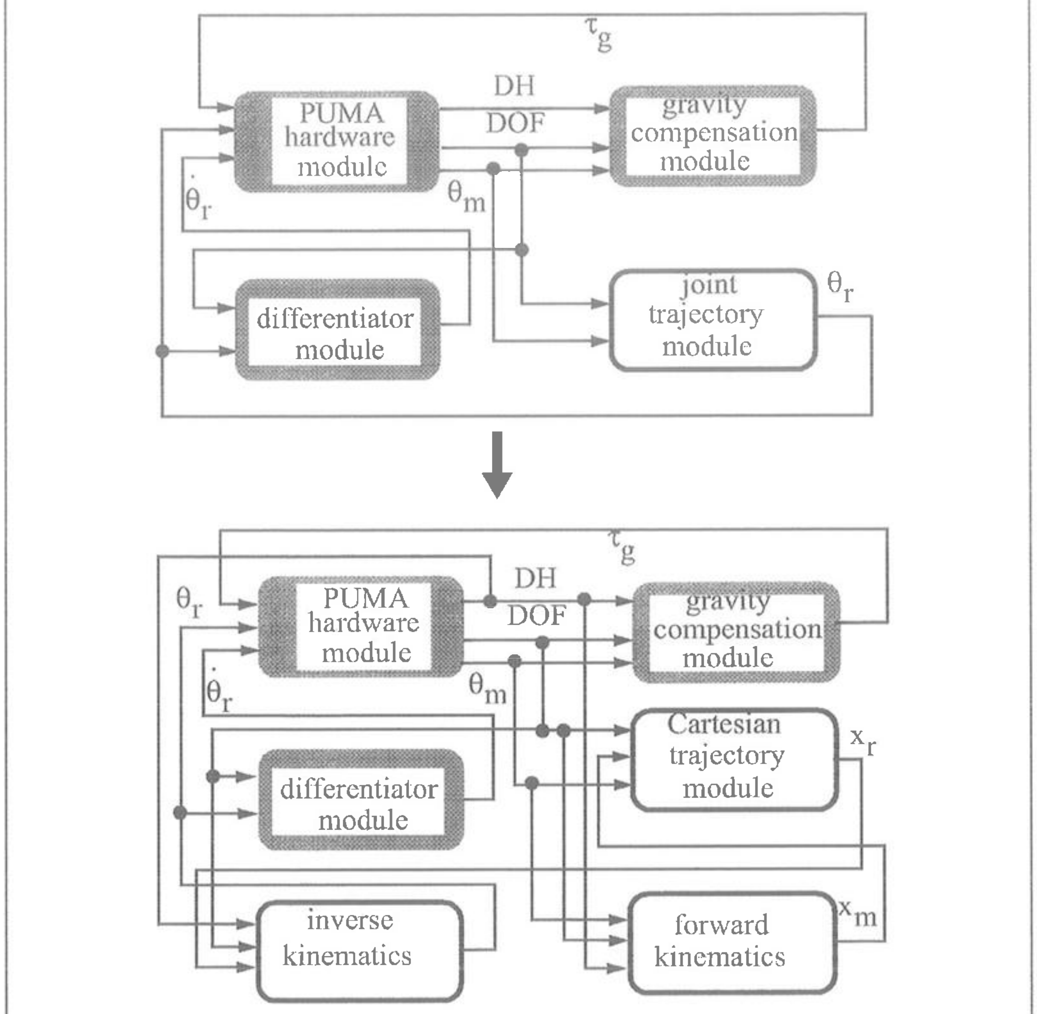


Figure 17: An example of subset dynamic reconfiguration from a joint position control configuration to a Cartesian control configuration. The shaded tasks are common to both configurations, and thus were not removed during reconfiguration.

deactivated on the RTOS and killed.[1] After this, the *TASK* nodes in T_2 which are elements of the union of T_1 and T_2 are updated with their current status on the RTOS, making their instances in T_1 superfluous. Next, the *CONFIG* node for T_1 is removed from the linked list of configurations and is replaced by that of T_2, and all nodes associated with T_1 are freed. Finally, the configuration window and library window are refreshed, with *CurrentConfig* being set to T_2 and with the proper tasks in the library "greyed out" in accordance with

1. The task set T2 - T1 could also be activated (i.e. cycled) between the deactivation and killing of T1 - T2; however, this section of code in Onika is currently commented out pending future work in the Chimera RTOS aimed at ensuring the proper activation order of tasks within a configuration.

Equation (2). It is important to note that the tasks which are elements of the union of T_1 and T_2 are generally not affected on the RTOS during this process in any way, except that they are now represented by *TASK* nodes in T_2 rather than T_1. An exception is the case when a task which has one or more *OUTCONST* SVAR is exchanged for another during reconfiguration. In this case, tasks with the corresponding *INCONST* SVARs are reinitialized automatically by Onika to get the new value(s), the process of which forcefully deactivates them. Onika subsequently reactivates these tasks automatically, so that all of this procedure is transparent to the user.

Superset dynamic reconfiguration requires foreknowledge of all configurations which will be used sequentially, such as occurs within a high-level Onika application. Within Onika, automatic superset reconfiguration is performed as follows: the union of the sets of tasks for each configuration is spawned at the beginning of the execution of the application, and the application is traversed by activating or deactivating tasks as required until the end of the application is reached, at which time all tasks are killed. Since the spawning of tasks only occurs at the beginning of the execution of the application, and are killed only at its termination, the overhead required during reconfiguration from one job to the next is minimal compared to that of subset dynamic configuration. The major drawback of superset dynamic configuration is that different tasks which require exclusive access to the same hardware cannot exist within the same application, since access to the hardware is typically attempted when the task is spawned, and therefore the two tasks would be in conflict. Superset dynamic reconfiguration is discussed further in Section 5.9.

5.3.5 Rendering of tasks and connections

The rendering of tasks in a configuration is identical to that of tasks within the library, except that visual feedback relevant to the task on the RTOS is given as well. This includes its state (*ON*, *OFF*, or *ERROR*) and the name of the RTPU on which it was spawned. This information is printed within the icon beneath its name. Additionally, if Onika is run on a system with a color monitor, then tasks which are *ON* (those either cycling or in synchronization) are colored green and tasks in *ERROR* are colored red. If Onika is not connected to the RTOS, then RTPU and state information are suppressed.

Connections between tasks within the configuration are performed automatically, relieving the user from the tedious and error-prone burden of manually connecting a pin on one task with those on other tasks. Each configuration's *CONFIG* node is associated with a linked list of *LOCUS* nodes, the structure of which is shown in Code 4. Each state variable referenced in the tasks of the configuration has one *LOCUS* node, which contains

```
typedef struct clocus {
    struct clocus *next,*prev;    /* For use in linked lists */
    NAME_STRING var_name;         /* The svar name of this locus */
    int amount,                   /* The number of tasks using it */
        locus_x,locus_y;          /* Its location on screen */
    int ic,                       /* Number of INCONSTs */
        oc,                       /* Number of OUTCONSTs */
        iv,                       /* Number of INVARS */
        ov,                       /* Number of OUTVARS */
        din;                      /* Number of double inputs */
    int locus_active;             /* Is a task supplying a value now? */
    int locus_show;               /* Should the conn be shown? */
    int locus_list;               /* Should the value be shown? */
    } LOCUS;
```

Code 4: The structure of a LOCUS node, used for automatic connections between tasks.

fields listing the total number of tasks which reference this state variable, the manner in which they are references (as *INCONST*, *OUTVAR*, etc.), and so forth. There is also an (x,y) coordinate pair associated with each connection locus which serves two purposes. First, it gives the user a place in the window to click in order to get information about a particular state variable. Second, it is used as a central location for all task pins referencing that SVAR to draw to. Therefore, tasks with SVARs in common do not literally connect graphically from one to the other; rather, the connection is made by routing a connection from either pin towards the central "locus," where the two meet. A built-in "route-around" algorithm keeps the connection lines from running across icons which might be located between a pin and its target locus.

When a task is added to a configuration, each of its pins are checked to determine whether or not they references SVARs which are do not already have nodes in the locus list. If none existed previously, a node is created, initialized, and inserted into the list, and the locus is assigned coordinates a few pixels horizontally outwards from the pin. When another task is added which references the same SVAR, the locus node is updated to reflect this information, and the locus position is moved to a location half-way between the two pins. If a locus is ever assigned a coordinate which would place it on top of any icon, the coordinates of the locus are updated 10 pixels horizontally and vertically until the locus is no longer atop an icon. Additional references to that SVAR by other tasks update the *LOCUS* node information, but do not change the location of the locus. If enough tasks are deleted so that only one task remains, the locus is assigned coordinates a few pixels outwards from the remaining referential pin. When the last task referencing the locus is deleted, then the *LOCUS* node is removed and freed.

Loci are "sticky;" that is, if a task is moved, the loci to which it is connected do not change position. The user can graphically drag the loci to different locations as desired.

Loci are rendered as filled-in circles of a user-defined size (the default is 3 pixels in radius). The configuration event loop checks for mouse events occurring to these loci, so that the user can use a mouse to get information as to the current value and task distribution of an SVAR, to toggle whether or not the connection is seen (the field *locus_show* is used as a flag for this purpose), and to drag the locus to a new location. Appendix B.2 details how mouse commands affect connection loci.

Connections are rendered in the following manner: starting with the first pin in the first task in the configuration linked list, Onika performs a list search to find the corresponding locus entry. When found, Onika checks to see whether the connection should be shown and, if so, what color it should be. SVARs for which an active task is generating output are drawn red (or solid for a monochrome monitor), whereas SVARs which are referenced only by input pins or inactive output pins are drawn blue (or dotted for a monochrome monitor). Finally, Onika plots a path from the pin to the locus which avoids any icon which might get in the way, using a simple "look ahead" condition which takes advantage of the fact that all icons are of the same width. Similar routing algorithms are currently existent in CAD software packages (such as those designed for PCB layout), but, to our knowledge, have never been used previously within a modular programming environment. The algorithm for our routing algorithm is given in Code 5. We do not mean to suggest that this algorithm is foolproof, as one can contrive situations where a connection would be forced to pass over an icon using this mechanism (for instance, when two icons are placed side by side with no space between them). Instead, we offer it as a "proof of concept," wherein we demonstrate that a human-machine interface can automatically connect icons graphically without human intervention given a reasonable algorithm to do so. A better algorithm (perhaps adapted from techniques used for PCB layout) would involve recursion to make certain that the "bypasses" themselves do not pass over icons, but this must be weighed against the overhead incurred by recursion in an environment where screen refreshes can be already be perceptibly measured in time.

Connections and loci are redrawn every time a task is placed, repositioned, moved, activated, or deactivated, or whenever a new configuration is viewed. Connections which the user has toggled to be "invisible" are not drawn; instead, the loci alone is drawn at its coordinates, with the name of the SVAR it represents printed beside it.

```
connect_to_locus(pin,icon,locus)
{
    if (pin is not on same side of icon as locus) {
       draw() out from pin horizontally;
       if (locus's y coordinate is above or below icon) {
          draw() to locus's y coordinate;
          draw() to locus's x coordinate;
       } else {
          draw() up a level a few pixels above icon's height;
          draw() to locus's x coordinate;
          draw() down to locus's y coordinate;
       }
    }
    else if (pin is on same side of icon as locus) {
       draw() to locus's x coordinate;
       draw() to locus's y coordinate;
    }
}

draw(x1,y1,x2,y2)       /* line will be horizontal or vertical */
{
    if (y1==y2) {       /* line is horizontal */
       rearrange points so that point 1's x is smaller than point 2's;
       xc=x1; step=ICONWIDTH/2;
       while (xc != x2) {
            if (xc+step > x2) line_to(xc=x2,y2);/* done */
            else if (no icon at (xc+step,y2))
               line_to(xc=xc+step,y2);          /* continue */
            else {                              /* route around icon */
               line_to(xc,above icon);
               line_to(xc=xc+step,above icon);
               line_to(xc,y2);
            }
       }
    else if (x1==x2) {  /* line will be vertical */
    {
         /* etc... */
    }
}
```

Code 5: The algorithms (in pseudocode) used for autoconnecting a pin on a task with the appropriate connection locus.

Pins on tasks within a configuration are connected on the basis on whether the names of the pins match or not. The assumption made, therefore, is that the user will want pins with identical names connected. Of course, one can postulate a case wherein two pins are erroneously connected; for instance, if a user forgets to alias a generic differentiator module's pins from (for example) measured joint positions to referenced joint positions. However, the convenience of not having to manually route a large number of connections far outweighs any problems encountered by the "autoconnection" routine (for instance, a simple joint motion configuration having only four tasks contains seventeen I/O pins referencing seven different state variables, all of which need to be interconnected).

5.3.6 Task manipulation and control

Tasks are manipulated in the configuration window in the same manner as those in the library window, with a few added features. Tasks are placed into a configuration by selecting them in the library window, moving the cursor to the configuration window, and pressing and releasing the right mouse button at the desired spot. If the configuration is connected to an RTOS, the user will be prompted for the RTPU upon which the task should be spawned (if more than one is available) before the task is rendered. Tasks are spawned on the RTOS in an inactive state. Placing a task causes a refreshing of the library and configuration windows.

The user can "flip" an icon so that the input pins are on the right side and outputs are on the left by selecting it and pressing the "R" key, as this may make a configuration more graphically pleasing to the user. When this is done, the arrow indicating input flow direction is reversed. Single-instance loci associated with the icon will also be moved appropriately. The configuration window is refreshed after an icon is reversed in this manner.

Tasks may be deleted and replaced in the configuration. To delete a task, the user selects it and presses the "Delete" key. This causes (after the user is asked "Are you sure?") the associated task on the RTOS to be deactivated and deleted, and excises the TASK node for the icon from the linked list. Connections are updated appropriately, and the library and configuration windows are refreshed. The icon is held in a buffer, and a copy may be pasted back into the program (subject to the normal rules regarding task interference) at the point from where it was deleted by pressing the "V" key, again refreshing and updating all windows. The icon need not be pasted back into the same configuration, but can also be pasted into a different configurations as well. The key command "C" causes a copy of the currently selected icon to be placed into the buffer without removing it from the current configuration. Only one icon may exist in the buffer at a time; an icon in the buffer is freed if another icon is deleted or copied.

The activation state of a task may be toggled if an RTOS session is in progress by selecting the task with the shift key down. If a task is *ON*, it is set of *OFF*; if *OFF*, it is set to *ON*; and, if in *ERROR*, Onika will ask the RTOS to clear the error and set the task to *OFF* if possible. If not possible, the task will remain in the *ERROR* state, but can still be deleted and replaced by the user. In all cases where toggling is successful, the configuration window is refreshed.

Certain tasks (generally trigger tasks) require additional user input before they can be activated. When the user attempts to turn these on, the RTOS will inform Onika that more information is needed. Onika formats this request and presents it to the user within another window. The user enters the information, and Onika determines whether or not it lies within the establish boundaries for acceptable answers. If so, the information is sent to the RTOS, and the task begins cycling; otherwise, the user is asked to re-enter the information.

If an RTOS session is underway, the RTOS will inform Onika when major events occur to the tasks; e.g., the task has turned itself off, the task has finished, the task encountered an error. Onika refreshes the screen to show the current state of the tasks whenever such a message is received.

Once a configuration of tasks has been tested thoroughly on the RTOS, the user may wish to save it as a high-level job. In the following sections, we discuss the creation and use of high-level icons.

5.4 Job Dictionaries

High-level job and target icons are stored in job dictionaries. Unlike task icons, the job and target icons are not created on the fly from a description file, but are in fact loaded in from files relating exactly how the icon should appear. Outside of that point, the process loading of icons into the dictionary is very similar to that of loading tasks into the task library.

Because the upper level of Onika is highly dependent on the use of color to identify icons, monochrome monitors are incapable of presenting the upper level in any aspect.

5.4.1 Loading jobs and targets

As mentioned in Section 5.2.1,Onika may be launched in such a way as to automatically search for a Chimera connection, or the user may choose to run an Onika session without

```
typedef struct onikon {
    struct onikon       *prev,
                        *next;
    NAME_STRING         name;     /* name of onikon */

    char                desc[MAXLINELEN];
    int                 type;     /* object, icon, etc. */
    UI                  *ui;      /* object data, branch data, etc. */

    int                 x,y;      /* location on workspace */
    byte                pict[ONIKONHEIGHT][ONIKONWIDTH];
                        /* the icon's picture */

    int                 line;     /* what line is it in */

    MARKER              marker;
    char                synchtag[MAXNAMELEN];
    char                whiletag[MAXNAMELEN];
    int                 whiletest;
    struct oniflow {
                        struct oniflow *prev,*next
                        struct onikon *info;
                        byte hole;
    } *flow;
} ONIKON;
```

Code 7: The structure of an Onika *ONIKON* node. The fields after and including *line* are used only when the icon is in an application.

any Chimera connections. In the prior case, Onika asks Chimera for the locations of desired task software dictionaries; in the latter case, Onika gets this information from the UNIX environmental variable *$ONIKA_LOCAL*. These locations may refer to directories on the local file system, or may refer to directories on remote file systems on the network (again, as discussed in Section 5.8.4).

Given the locations of these dictionaries, Onika searches them for files with the suffix *.onk*. These files contain all of the information necessary to draw a high-level icon, and also include default information for target icons. An example of such a file is given in Code 6. Starting with the first *.onk* file in the first dictionary, Onika loads the information in this file into a *ONIKON* node (shown in Code 2). The node is added to a double-linked list of *ONIKON* nodes for a given dictionary, which is then sorted (using a selection-sort) so that icons of the same type are grouped together. Each software dictionary opened has

```
VERSION     1.38
ONIKON      Joint object
DESC        above square's origin
UI          -0.17 -0.453 2.92 -0.0371 0.717 1.29
UI          7
PICT        88888888888888888888888888888888
PICT        87777777777777777777777777777778
PICT        87777887788877788778778788887778
PICT        87778778787787877878778787777778
PICT        87778888788877787787878788877778
PICT        87778778787787877877878787777778
PICT        87778778788877788777787788887778
PICT        87777777777777777777777777777778
PICT        87777777777777777777777777777778
PICT        87777777777777777777777777777778
PICT        87777777777777777777777777777778
PICT        87777777777222222222277777777778
PICT        87777777772222222222227777777778
PICT        87777777772222222222227777777778
PICT        87777777772222222222227777777778
PICT        87777777772222222222227777777778
PICT        87777777772222222222227777777778
PICT        87777777772222222222227777777778
PICT        87777777772222222222227777777778
PICT        87777777777887777778877777777778
PICT        87777777777777777777777777777778
PICT        87777777777777777777777777777778
PICT        88888888888888888888888888888888
PICT        88888888888888888888888888888888
PICT        88888888888881111118888888888888
PICT        88888888888881888818888888888888
PICT        88888888888881888818888888888888
PICT        88888888888881888818888888888888
PICT        88888888888881888818888888888888
PICT        88888888888881111118888888888888
PICT        88888888888888888888888888888888
PICT        88888888888888888888888888888888
EOF
```

Code 6: An example target icon's *.onk* file.

its own linked list of nodes. The dictionary information, in turn, is stored in a node and entered into a double-linked list of dictionary nodes. (Dictionary nodes are the same as *APPL* nodes, described in Section 5.5.)

All target and job icons must have unique names. If the name of an icon matches that of

Page 94 of 198

ABBINC_0125850

another icon already loaded into any dictionary list, the icon will not be added to the current dictionary list.

When all icons have been loaded, Onika refreshes the dictionary window, thereby causing the icons to be drawn. Only one dictionary can be seen in the window at a time; this is the dictionary pointed to by the global variable *CurrentDictionary*. The user can view different dictionaries at will by selecting the appropriate dictionary button at the top of the window; this has the effect of changing the value of *CurrentDictionary*. To present a dictionary visually, Onika traverses the dictionary's linked list of task nodes, starting at the first node. First, the appropriate edges are drawn for the icon and filled with the appropriate colors, based on its *type* field. Note that the edges and colors of an upper level icon are not specified in the icon's *.onk* file directly, but are implied by the icon's type as specified in the *ONIKON* field of the file. This makes icons portable to other sites even if other edge/color combinations are used to specify icon syntax.

After the edges have been drawn, the picture itself is drawn pixel by pixel, framed between its already-drawn edges. All lines for the edge are drawn thickly if the icon is "selected." Finally, the icon's description is drawn centered beneath the icon. The average drawing time per icon is approximately 0.125 seconds on a Sparc 10. The refresh routine redraws the icons in the currently displayed dictionary every time an icon is selected or deselected.

As with the library and configuration windows, an "infinite" event loop assigned to the dictionary window catches all mouse events generated by the user. These events are evaluated, and those found meaningful are processed appropriately (for instance, clicking an icon with the left mouse button "selects" that icon). Appendix B.3 describes which mouse events have meaning within the dictionary window.

5.4.2 Manipulating icons

Icons can be manipulated in many ways. They can be selected, dragged, and modified, and their files can be viewed and edited. The selecting and dragging of icons is done precisely as in the case of task icons (Section 5.2.2). Additionally, by selecting a job icon with the *Shift* key depressed, the user can automatically load in the configuration it represents (unless the icon is in fact a nested application, in which case the appropriate application is loaded in instead). This is referred to as *expanding* an icon.

There are two ways to modify an icon. Both use the middle mouse button, with either no key or the *Shift* key depressed. In the first case, the user is presented with a window in

```
typedef struct appl {
    struct appl *prev, *next;
    CANV_STRING name;
    DIR_STRING dir;
    ONIFLOW *flow_header[MAXFLOW];
    ONIFLOW *flow_tailer[MAXFLOW];
    ONIKON *icon_header[MAXFLOW];
    ONIKON *icon_tailer[MAXFLOW];
    ONIKON *selected;          /* The currently selected icon */
    Panel_item button;         /* Selects this application */
    int numflow;               /* Number of flow lines */
    int parallel;              /* Parallel or conditional? */
    byte dirty;                /* Has application changed? */
    char *alias[MAXFLOW];      /* List of module substitutions */
    int aliasnum[MAXFLOW];     /* Number of substitutions */
    int lines[MAXFLOW];        /* Flow positions (wrap-around?) */
} APPL;
```

Code 8: The structure of application nodes. When used as a dictionary node, the fields *flow_header*, *flow_tailer*, *parallel*, *dirty*, *alias*, *aliasnum*, and *lines* are unused, and *numflow* is always 1.

which the description of the icon (and, in the case of target icons, the data of the icon) can be changed. In the latter case, the user can modify the appearance of an icon by bringing up the icon creation window of Section 5.6.1. Modifications to icons affect only the icon itself and any instance of the icon created from it since it was changed. Icons already existing in applications are not affected.

5.5 Applications

In order to use the icons in the dictionary to control the real-time control system, the icons must first be placed into an application. This is done by selecting an icon in the job dictionary, moving the mouse over the icon after which the new icon is supposed to follow, and pressing the right mouse button. Unlike the placement of task icons into a configuration, the location of the newly-placed icon is directly relevant to its performance during execution of the application. This is because high-level applications are goal-oriented, and their component icons form a "story" which is executed in a certain order, one job after another. This is unlike the lower-level case, in which icons represent parallel-executing tasks all simultaneously cycling. In this section, we further describe the higher-level applications.

5.5.1 Application formats

The format for application nodes is given in Code 8. Application nodes are maintained in a double-linked list. They are also used as dictionary nodes, but in such cases many of the fields are unused. Only one application is viewed at any given time (pointed to by *Curren-*

```
typedef struct oniflow {
    struct oniflow *prev,*next;
    ONIKON *info;
    byte hole;                    /* Flags a gap between icons */
} ONIFLOW;
```

Code 9: The structure of an *ONIFLOW* node.

tAppl). Similar to that of the library, configuration, and dictionary windows, the user can select the desired application for viewing from a row of buttons at the top of the window, changing the value of *CurrentAppl* and causing the application window to be refreshed. Unlike the lower-level configuration window, no application window needs to be open at any given time; applications do not represent the "status quo" of the RTOS, but rather a desired series of steps and changes to be performed on one or more subsystems at run-time. The user may create new applications, or can load in pre-saved applications (the file format for applications is shown in Appendix A.4). Applications may have multiple flows, which may represent parallel lines of execution, or different conditional branches (the exact nature of the application is shown in the information section of the application window). All application flows have a minimum of two icons in them, signifying the start and end of the flow. The existence of multiple flows requires special application traversal methods when creating and executing an application; this is dealt with in the next section.

5.5.2 Application traversal

The *ONIKON* nodes of an application are simply appended to the application's linked list as they are inserted into the application, and their ordering in that list does not reflect their position in the application itself. Instead, a linked list of *ONIFLOW* nodes (shown in Code 9) contain the correct ordering for the application. Each *ONIFLOW* node has a pointer field which can reference an *ONIKON* node, and vice-versa (shown in Figure 18). The *ONIFLOW* nodes are very small, and can easily be swapped around as needed, requiring much less time to do so than the much larger *ONIKON* nodes. Nevertheless, the *ONIFLOW* nodes would seem to be superfluous, since we could easily have eliminated them and simply caused *ONIKON* nodes to be inserted into the proper place in a linked list rather than appended to the list. Why, then, should there be a separate list relating the ordering of the icons?

The answer to this question lies with future expansion of Onika to support conditional applications in a more robust manner. Currently, a conditional application is supported by creating an application with more than one flow (thus having several separate linked lists

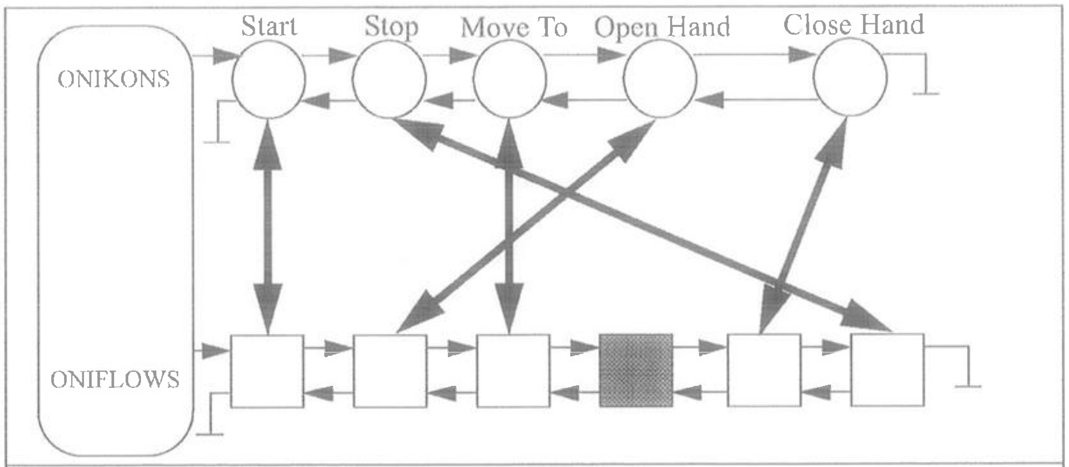


Figure 18: The *ONIKON* nodes for an application are linked in the order in which they were placed into the application, regardless of their location in the application. The *ONIFLOW* nodes are smaller and kept in the order in which they will be executed (i.e. the order that the user sees pictured in the application), and include blank nodes (such as the shaded one in this figure) to indicate places where "holes" occur.

of *ONIKON* nodes), iconifying this application, and placing it into a higher-level application (pictured in Figure 19). The return value of the job executing before the iconified application determines which flow of the iconified application will be followed. This is a brute force implementation of conditionals, inconvenient in that the entire conditional block must be viewed as a single icon in the higher-level application. Future developments of Onika will address this shortcoming by permitting graphical branching within an application. Application flows will therefore not be linear, but will be complex graph constructs wherein icons will continuously need to be moved, resized, shuffled, and so on (Figure 20). The use of *ONIFLOW* nodes anticipates this development by providing a simple and elegant way to traverse applications without major modifications to the *ONIKON* nodes and the procedures which manipulate them; only minor modifications to the *ONIFLOW* nodes and the procedures which use them will be required.

5.5.3 Laws governing the placement of icons

The rules governing the placement of high-level icons into applications are much less complicated than those which govern the placement of tasks into a configuration. Basically, an icon b may be inserted after an icon a if and only if:

$$L(T(b)) = R(T(a)) \qquad (7)$$

where the function $T(x)$ returns the type of icon x (e.g. *Joint job*, etc.) and functions $L(t)$

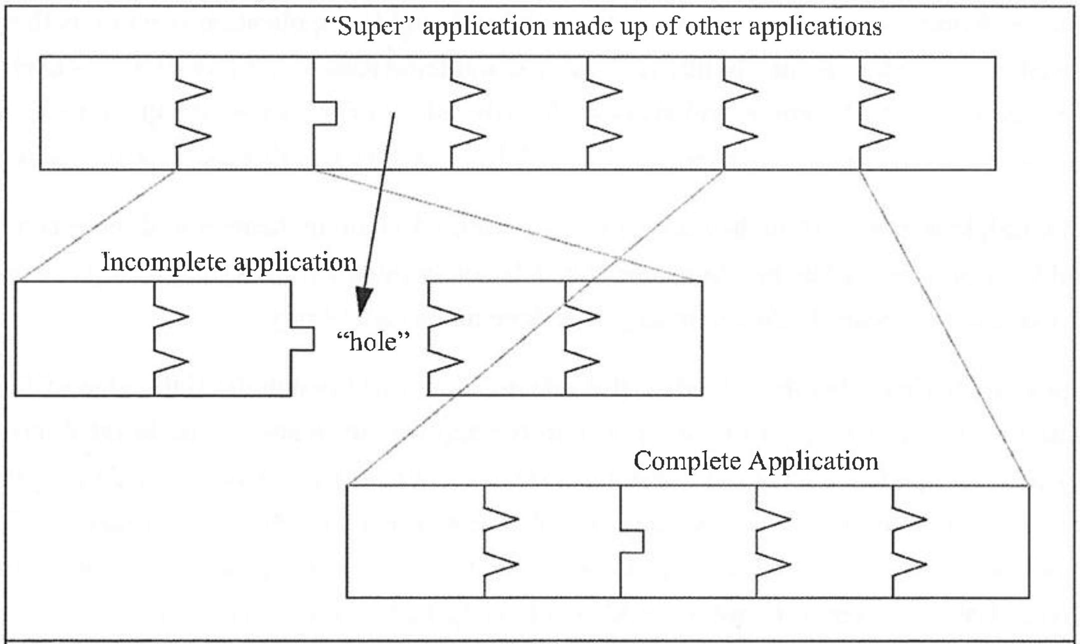


Figure 19: Two applications have been included in a "super" application. One is incomplete, and requires that the "super" application complete it with an object when it is used. The other application is complete, and requires no object when used.

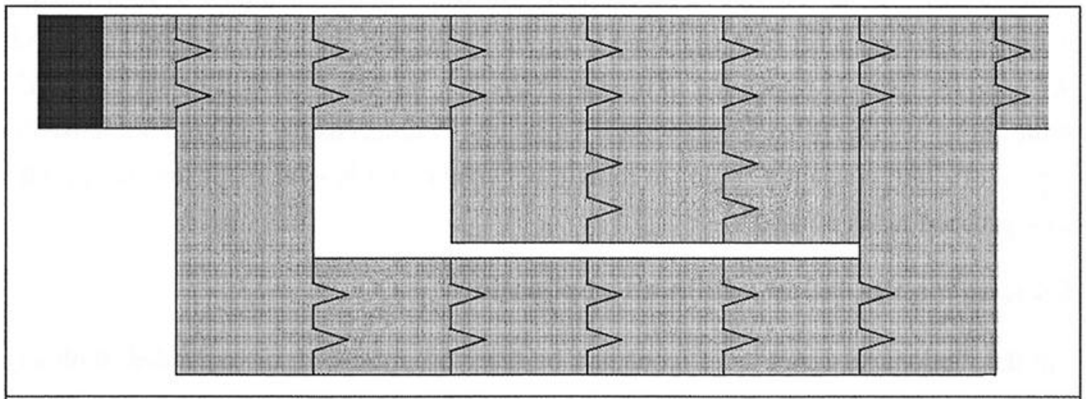


Figure 20: The use of *ONIFLOW* nodes anticipates the future development of graphically representing conditionals as a graph, wherein fast, concise routines will be needed to traverse and manipulate icons within a flow.

and *R(t)* return the left edge information and right edge information of an icon of type *t*, respectively. If Equation (7) is true, then an *ONIKON* node for *b* is created and appended to the list of *ONIKONs* for the flow, an *ONIFLOW* node is created for *b* and inserted in the proper place in the flow list, and the two nodes are set to point at each other. The coordinates of *b* and all icons which follow it are updated, and the application window is then refreshed to display the newly placed icon. The implementation of Equation (7) is made obvious to the user by color- and shape-coding the edges of the icons; if Equation (7) is true, then the edges of the icons in question will be seen to fit together like puzzle pieces.

As multiple instances of high-level icons are permitted in an application and the acceptability of an icon within an application depends solely on where it is placed, there is no "greying out" of icons in the dictionary as is done in the task library.

Equation (7) states that the left edge of a selected icon must match the right edge of the icon it is to follow if it is to be inserted into the application. Icons can be inserted anywhere in the application, subject to that rule. However, it may happen that it would be syntactically incorrect for the new icon to precede the icon which follows it. If Equation (7) fails when *a* is the new icon and *b* is the icon which would follow it when *a* was inserted, then a "hole" is inserted between *a* and *b* indicating that another icon (or icons) will be required to bridge the gap between them Figure 13 on page 37. This involves the additional creation and insertion of an *ONIFLOW* node between the *ONIFLOW* nodes of *a* and *b*. This new node will not point at an *ONIKON* node, and its *hole* field will be set to *TRUE*. The coordinates of *b* and all icons which follow it in the flow are updated to make room for this "hole."[1] If an attempt is subsequently made to place an icon *c* after *a*, then if Equation (7) succeeds for both (*a*,*c*) and (*c*,*b*), the "hole" will be made to point at an *ONIKON* node for *c* newly appended to the *ONIKON* list, its *hole* field will be set to *FALSE*, and the "hole" will be filled. Otherwise, if Equation (7) succeeds for (*a*,*c*) but not (*c*,*b*), the insertion of *c* will cause the "hole" to remain between *c* and *b*, with the insertion of *c* otherwise proceeding as normal.

5.5.4 Icon manipulation within an application

As in the dictionary, upper level icons can be selected, modified, or expanded, with any modification affecting only the currently selected icon instance (Section 5.4.2). Unlike the dictionary, icons cannot be dragged, since they exist within a highly-structured goal-oriented application; however, they can be deleted, copied, and pasted. Deleting a selected icon (using the *Delete* key) removes it from the list of *ONIKONs* and *ONIFLOWs* for the application flow (using Equation (7) and the creation or deletion of "holes" to reposition

1. We could as easily insert a hole between a new icon and the icon after which it should follow if Equation (7) fails, rather than forbidding the insertion; in fact, the code to do this has existed in Onika since the early upper-level prototypes, but is currently not activated. This was to prevent the gratuitous creation of holes when the user accidentally tried to place (for instance) a Cartesian target icon after a joint job icon.

the remaining icons as necessary), and causes its *ONIKON* node to be stored in a buffer. A copy of the currently selected icon can also be stored in this buffer without actually deleting the icon by pressing the "C" key. Only one icon can be stored in the buffer; an existent node in the buffer is destroyed in favor of newly copied or deleted nodes. A node in the buffer may be pasted after the currently selected icon (subject to Equation (7)) by pressing the "V" key. Icons in the buffer may be pasted multiple times, within the same application or into a different application.

Additionally, an icon may be marked as being part of a structural command. There are three such structural commands within Onika; *breakpoints*, *synchronization points*, and *loop delimiters*.

If a job icon is marked as a breakpoint (toggled by selecting the icon and pressing the "M" key), the execution of the application will pause after the job is executed. If the icon is part of a job-target tandem, then it is unimportant which icon in the pair is selected and marked; toggling either will be effective, and only one "mark" is held for the entire action.

By selecting an icon and pressing the "S" key, the user can assign a synchronization label to the icon (or action, if the icon is part of a job-target tandem). Synchronization tags are used only for parallel applications. If an action in one parallel flow has a certain synchronization tag, then Onika will not cause it to execute until the same synchronization tag is encountered in a different flow. This guarantees that two jobs on different subsystems will begin execution simultaneously (inasmuch as this is possible using a single-threaded interface and controller). This feature which, to our knowledge, has never been implemented in any other real-time control interface, eliminates the need for bracketing sets of instructions with "cobegins-coends" or the need to calculate pause times in an attempt to get the starting times of jobs to be the "same," all without requiring any communication between the parallel subsystems.

Pressing the "W" key allows the user to assign a loop label to the currently selected icon. This is similar to the assigning of synchronization labels to icons, except that both instance of the label must be within the same flow of the application. One instance of the loop is designated the "test" icon, whereas the other is the "anchor" icon. If the return value of the test icon's job indicates that looping should occur, then execution continues at the anchor icon, regardless of whether it is to be found before or after the test icon. Thus, both bottom-test loops and top-test loops are supported.

Icons which are part of a programming structure have special rendering requirements. The rendering of icons in applications is the subject of the next section.

5.5.5 Rendering of icons in applications

Icons in an application are rendered in much the same manner as those in the dictionary, with a few exceptions. First, the descriptions of the icons are not printed underneath the icons, since the available space prohibits this. Next, certain icons will have been marked as breakpoints, synchronization points, or as part of a top- or bottom-test loop. Finally, there may be wrap-around in flows which are very long.

Icons which are marked as breakpoints have a small blue triangle drawn point-down above the upper right corner of the icon (or, if part of a job-target tandem, above the upper right corner of the target icon). Synchronized icons have their labels written above them (above the job icon in a job-target tandem). Icons which define the starting and ending point of a loop have the loop label drawn beneath them, with an arrow drawn from the "test" icon to the "anchor" icon indicating the flow of execution if the loop is taken.

The array *lines* within an *APPL* node reports how many "lines" a particular flow within an application requires due to wrap-around. This supports a positional updating routine within Onika which causes (for instance) the second flow of an application to automatically be drawn slightly lower within a window if the first flow line wraps around one or more times. An arrow is drawn to illustrate the flow of code if wraparound occurs.

In the next section, we discuss how the icons used in the upper level of Onika are created from lower-level software and included in the software dictionaries.

5.6 Creating and Representing Icons

In order to create jobs from configurations, to create targets for these jobs, and to use previously created applications within other application, we have developed an icon creator and modifier for Onika. An important and unique feature of this icon creator is the ability to automatically determine and assign the syntax of a newly created job or application icon. In this section, we discuss the icon creator and its functionality in more detail.

5.6.1 Icon creation and modification

The icon creator and modifier developed for Onika is a simple pixelmap editor which presents a grid of the size *ONIKON_WIDTH* x *ONIKON_HEIGHT* to the user into which a representative icon can be drawn. Each grid element is 10 x 10 pixels, and can be any one of nine colors (black, white, grey, red, green, blue, yellow, green, and orange), with the exception of the elements on the border, which are always black. Onika reserves only nine colors for icons to minimize color table usage and the "flash" one experiences when shift-ing between programs with different color tables; however, this is an arbitrary number chosen to give a wide-enough choice to icon designers while simultaneously demonstrating icon creation, and could certainly be modified in future releases.

The icon creator window is made visible when the user attempts to save a configuration as a job, when the user wishes to create a new target, or when the user wishes to modify the image of some pre-existing icon in an application or in the job dictionary. The icon creator window is non-blocking and does not interfere with the flow of events in other Onika windows.

The icon creator window is divided into three sections: the grid, the view panel, and the control panel. The grid, described in the previous paragraph, is manipulated in much the same manner as are images in any painting package; the color of the pen is selected from a palette, and the user can draw lines, rectangles, or circles, can draw freehand, or can fill in areas with a certain color. Only the interior of the icon is designed by the user; the colored/shaped edges of the icon are automatically determined and assigned by Onika, based on the type of icon being created. The view panel shows an "actual size" image of the current icon under development, and the control panel contains the palette, the drawing mode (pen, line, circle, etc.), the *Save/Cancel* buttons, the *Clear* button, and the *Undo* button, as well as information about the name of the icon and its type.

Onika keeps two copies of the grid in its memory: the current picture, and the picture before the last event. When a drawing event occurs, the resulting picture is copied to the non-current buffer, which is then declared to be "current." Using the *Undo* button causes Onika to make the alternate picture buffer "current." This allows for a speedy "undo" and "redo" of any picture event, since there is no swapping of memory involved, but instead only the reassignment of a pointer to the alternate buffer.

When the user decides that the picture is completed, then the *Save* button can be pressed to write the picture to the appropriate window and file.

5.6.2 Job representation

Several steps are involved in creating a job icon from a configuration of tasks. In this section, we detail the iconification process.

When a user requests that a configuration of tasks should be saved as a high-level icon, Onika first ensures that the configuration has been saved since any changes were made to it, and then analyzes the configuration to see if it contains any hanging inputs, using Equation (6). If hanging inputs exist, and if the user has specified that Onika should issue warnings whenever suspicious configurations occur, then the user is informed as to which inputs are hanging, and is given a chance to gracefully cancel the iconification process.

Next, the configuration is scanned to determine whether or not it contains any trigger tasks (Section 3.2.2). If the configuration contains a trigger task (for example, a joint trajectory generator), then the iconified job will be assigned the type listed in the *OBJECT* field of the trigger task's *.rmod* file (for instance, *Joint job*) to indicate that this will be a job requiring a target object. Currently, Onika only checks for the existence of a single target task; if, for some reason, multiple trigger tasks existed within the configuration, the first such task encountered in the linked list of tasks would define the job type, with the remainder being ignored. If no trigger task is encountered, then the default job type is *Complete job*; i.e. a job which requires no target icon to follow it in an application.

After the job type has been determined, and the job name has been assigned (the name of a job icon is the same as that of the configuration which it represents), then the user is presented with the icon creator window discussed in Section 5.6.1. The user creates a suitable image for the icon, and presses the *Save* button (or *Cancel* to abort the entire process gracefully). The job icon is then saved in a *.onk* file (the format for this file is given in Appendix A.3).

Finally, an *ONIKON* node is generated for the job, which is then appended to a dictionary linked list. The exact destination dictionary is determined as follows: if the configuration the job represents is (for instance) */usr/mwgertz/base/conf/myjob.conf*, then the resulting job file will be */usr/mwgertz/base/dict/myjob.onk*, and the icon will be added to the dictionary representing the */usr/mwgertz/base/dict* directory. If the affected dictionary is the current dictionary, then the dictionary window will be refreshed to display this icon.

5.6.3 Target representation

When the user wishes to create a target, he or she is presented with a listed of object types from which to choose. This list is generated from the *OnikaUpperSetUp.dta* preference file, which specifies the types of icons that Onika recognizes. (This will be discussed further in Section 5.7.) The user selects the desired type of icon from this list of buttons, after which the icon creation window of Section 5.6.1. is displayed. After creating a picture for the icon, the user is prompted for default values relevant to the specific target type. Finally, the user saves the new icon to a *.onk* file (the format for this file is given in Appendix A.3). If the location of the target's file lies in the province of an open dictionary, then an *ONIKON* node is created for it, and the dictionary window is refreshed to show the new icon if the appropriate dictionary is currently displayed.

5.6.4 Application representation

Applications themselves can be iconified and stored back into a dictionary, for use in higher-level application. The application need not be complete, as the resulting icon can be considered as an "application requiring an object". This section describes how application icons are created.

When the user requests that an application be iconified, Onika first ensures that the application has been saved, as the icon will need to refer to the application file during execution (the format for an application file, suffixed *.appl*, is given in Appendix A.4). Next, Onika checks the application to determine what kind of icon it will become. If the application is complete (that is, if it has no "holes" in it), it is considered a "Complete application" which will not require an object when used in an another application. If, however, the application has any target "hole" or "holes" in it, Onika will check to see if these holes will all accept the same type of target. If so, the new icon will be saved as an "application requiring an *xxx* target," where *xxx* specifies the type of target. When this icon is later used in an application, the target icon which follows it will be used to fill up all holes in the application (and any subapplications) represented by this icon.

If the application has a hole where a job should be, or if more than one type of target hole exists in the application, then it cannot be saved as an icon, and the process aborts gracefully.

Once the icon type has been determined by Onika, the remainder of the process involving the creation of a picture, saving the *.onk* file, and insertion into a dictionary proceeds exactly as in the case where a job icon is created from a configuration (Section 5.6.2). Parallel applications cannot be iconified, to eliminate the possibility of nested parallel subsystems within other parallel subsystems (currently unsupportable in the Chimera RTOS); conditional applications must be in order to function correctly. A conditional application which is run separately at the uppermost level (i.e. not as a part of another application) will follow the first conditional flow by default during execution.

5.7 User Preferences

User interfaces are often designed with a particular type of user or a particular scope of application in mind. Once the use of the interface occurs outside of these boundaries, however, the user performance may become severely impeded. To make the scope of the Onika VLE less susceptible to such failings, we designed it to present different (but interacting) interfaces for the different levels of users. Nevertheless, a users performance may be affected by seemingly innocuous variables such as the color of icons and the types of unit measurements used. In this section, we discuss how user preferences are implemented within Onika.

5.7.1 Syntax preferences

In the course of this dissertation, we have alluded to various types of high-level icons, such as "Cartesian job," "Joint target," "Complete application," and so forth. These icon types are not hard-coded into Onika, but are read from a preference file called *OnikaUpperSetUp.dta*, located in the Onika home directory. The structure of this file is given in Appendix A.7. Icon types are specified by a name, a left edge index, and a right edge index, as well as listing of data types, sizes, and descriptions if the icon is a target icon. Edge types (which the icon type indices reference) contain the index number, the color of the edge (in RGB notation), and information which describes the shape of the edge (a series of (x,y) coordinate pairs indicating cumulatively relative offsets from the top of the outer boundary of the edge). The ordering of the different types and edges are unimportant within their respective groups.

Several icon types have special meaning within Onika, and should not be removed from the preference file (although it is perfectly acceptable to alter the colors and shapes of the edges which they reference). These include *Application start*, *Application end*, *Complete job*, *Complete application*, and *Complete NOP. Application start* and *Application end* are the icon types for the first and final icons in an application, which are nonfunctional; if the *type* field of an icon matches the index value of *Application end,* for instance, Onika "knows" that the icon cannot be expanded, executed, or deleted, and will not attempt to do so. An icon of the type *Complete job*, when encountered during execution, will be expanded to the configuration it represents; when the job is finished, execution will continue with the icon immediately following it. Icons of type *Complete application* are similar, except that they are expanded to the application they represent; when this sub-application finishes, execution continues with the icon immediately following it in the superapplication. For *Complete NOP* icons, no executory operation is performed on the icon, but control immediately proceeds to the icon following it. (*Complete NOP* icons are used to "space apart" icons in an application to make the application more readable. When relevant, they always "return" a *TRUE* signal for looping and a *GO_TO_FLOW(0)* signal for conditional applications which may follow it.) In general, if the icon type contains the word "Complete," then Onika assumes that it will never be used in a job-target tandem, and will update the "current job" pointer appropriately after the icon's routine completes its execution. Since upper level routines in Onika are either jobs or applications exclusively, there should be no need to add to the number of "Complete" icon types.

If an icon type contains the word "job" or "application," but not the word "Complete," then Onika will assume that the icon type represents an action which requires a target. Similarly, if the icon type contains the word "object," then Onika will assume that the icon type represents a target. During execution of an application, Onika uses these assumptions to correctly navigate through an application. There is no limit to the number of job types or target types that can be created. Generally speaking, if the icon type *Xxx job* is added to the preference file, then *Xxx application* should be added as well, and should be otherwise identical in all respects to *Xxx job* in its parameters. (The only difference between the two types is that Onika will try to expand *Xxx application* into an application rather than a job configuration during execution.) The icon type *Xxx object* should be added as well, with its left edge and right edge defined to be the reverse of *Xxx job/application*. The names are used for parsing when determining the type of job or application an icon should be (i.e. if an icon of the type *Joint object* is missing from an application, then the application will be saved, if iconified, as a *Joint application*), but Equation (7) uses the edge indices of the icon types when checking syntax, not the names, so that an icon of type *Xxx job* could be followed by an target icon of type *YYY object* if the left edge of the latter was the same index value as the right edge of the former. All of this, of course, is transparent to the typical Onika user, who simply matches colors and edges in puzzle-fashion, so care must be taken by the Onika maintainer not to create situations where incompatible jobs and targets are assigned edge indices allowing them to erroneously satisfy Equation (7).

The user has the option of creating his or her own preference file. If the file *~/.OnikaUpperSetUp.dta* exists, then Onika will use it in instead of the main preference file. This allows individual users to create their own icon types. However, there is no mechanism in Onika to address cases where an icon in a dictionary is of a type not recognized by the currently used preference file. Job and application icons otherwise not listed in the preference file are treated and displayed as *Complete job* or *Complete application*, whereas unrecognizable targets are simply ignored and not included in the dictionary.[1]

5.7.2 Basic preferences

Each individual user has the ability to specify certain preferences to be in effect for his or

1. This does not mean that the job or application icon is totally unusable. Instead of getting the target information required from some following target icon, the user is prompted for the information when needed by Onika, assuming the underlying tasks have been created according to established guidelines. This is true for all job configurations where a task requires input but is not a recognized "target task."

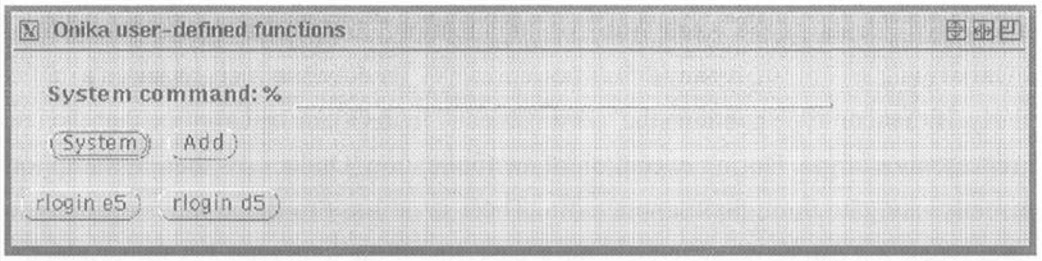


Figure 21: The user can define frequently-used system calls as buttons to be used from within Onika.

her session. These preferences do not affect the recall or storage of routines in any way, thereby not impeding the sharing of software. Included in this list of preferences are the size of task icons, the length of I/O pins, the number of characters per line for the names of icons, the amount of separation between connections in pixels, whether connections of various types are displayed or not, whether periods should be shown instead of frequencies, and so forth. These are all stored (along with the preferred locations and sizes of all Onika windows) in the file *~/.onika*.

There also exists a window in Onika in which the user can create buttons and associate them with system calls, which will be executed in the background when their buttons are pressed (Figure 21). The user's preferred buttons are stored in a file called *~/.onika.buttons*.

5.7.3 Additional preferences

The "Start of application" and "End of application" pictures can be changed from their normal start/stoplight images by editing the files *OnikaStart.dta* and *OnikaStop.dta*. in the Onika home directory. This is useful for cultures where the normal images would be meaningless. Changing these images will not affect the portability nor executability of any application in which they are a part.

All text statements in Onika are defined in the file *OnikaStrings.h*. To create a version of Onika in which all of the textual statements were in some other language (for example, Spanish), only that one file would need to be changed before recompilation of Onika. This functionality was inspired by the Apple User Interface and Compatibility Guidelines, which have been shown to be very effective when used for porting code between different cultures [35].

5.8 Communication

Onika communicates with its supporting real-time operating system via the Internet, and also uses the Internet to retrieve software not found locally. Furthermore, external subsystems required by tasks (such as real-time displays and path planners) are also supported via Internet sockets, although not directly through Onika. In this section, we present the various communication mechanisms through which Onika interacts with its environment.

5.8.1 The ENET socket package

To facilitate Internet communication between Onika and its supporting real-time operating system, we have developed a socket package (called the ENET socket package) designed for ease of use by programmers. This socket library includes the following commands: *enetCreate()*, *enetAttach()*, *enetSend()*, *enetReceive()*, *enetDetach()*, and *enetDestroy()*. These commands were incorporated into the Chimera RTOS principally for communication with Onika, but they are also useful for any general communication between two processes.

The procedure *enetCreate()*, used by the server process, takes as input the name of the socket connection, which is an arbitrary string of characters (except that the character '@' may not be used, for reasons discussed below). This string is separated into 4-byte pieces (padding the final piece if necessary), which are added together to create a 32-bit integer which is subsequently used as a socket file descriptor (SFD). A socket is created using this SFD, and the process blocks on a connection to that socket. A structure containing the SFD is returned if the procedure is successful.

The procedure *enetAttach()*, used by the client process, also takes as input the name of the socket connection, but this name may include a machine IP address or name appended to it as well, separated by an '@' character. For example, if the server on machine *A* created a socket using the call *enetCreate(mysocket)*, then a client process on machine *B* would connect to it by issuing the command *enetAttach(mysocket@A)*. The argument to *enetAttach()* is separated into the socket name and the machine name. (If there is no machine name given, then the assumption is made that the client process is executing on the same machine as the server.) The socket name is parsed in the same manner as *enetCreate()* to return the SFD of a socket, and then standard TCP/IP socket calls are used to connect the client to the socket SFD on the remote machine specified by the machine name. If the connection cannot be made, the procedure aborts gracefully. A structure containing the SFD is returned if the procedure is successful.

The functions *enetSend(SFD,type,size,buffer)* and *enetReceive(SFD,&type,&size,buffer)* are similar to (and in fact use) the standard Unix *write()* and *read()* commands. However, data transmitted by *enetSend()* is broken up into 1 Kb sections and sent sequentially. This minimizes data loss when large amounts of information need to be sent across a network. The size of the entire data package is sent in the second byte of the first section, so that the *enetReceive()* will continue to read socket data for a single transmission until all sections of the message have been received. The first byte of the message contains an index to the type of message being sent (e.g. error notification, acknowledgment, etc.), so that the receiving process can use this value to process the message properly. The function *enetReceive()*, when called, blocks the process until a message is received.

The functions *enetDetach()* and *enetDestroy()* halt communication and cause the socket to be destroyed by the client and the server, respectively. (Processes should first send a message indicating that communication is to be terminated before issuing these commands, so that the other process can eliminate its own socket gracefully.) If a new connection is desired, *enetAttach()* and *enetCreate()* commands must be reissued. A server generally issues an *enetCreate()* command after an *enetDestroy()* command, in order to await another connection.

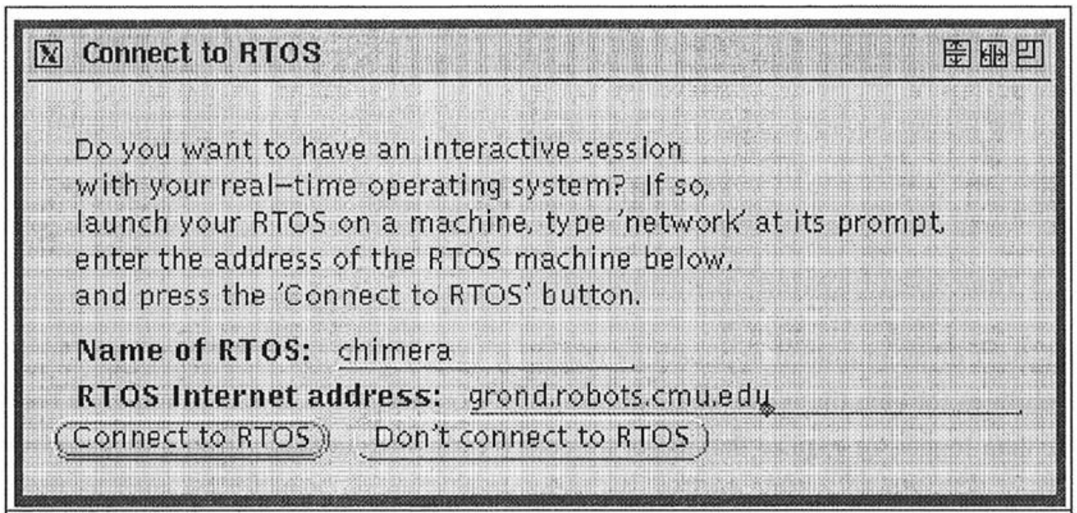
The ENET socket package is used for communication between Onika and the RTOS as well as between tasks and external subsystems. In the next section we discuss how this package is used to implement communication between Onika and the Chimera RTOS.

5.8.2 Onika-Chimera RTOS communication

To connect Onika to a Chimera RTOS session, the user must first execute a Chimera program containing a *sbsNetwork(socket_name)* call. This call causes Chimera to issue the command *enetCreate(socket_name)*, and informs Chimera that further commands will be issued via socket until the socket becomes inactive.

Onika issues an *enetAttach()* command either at start-up (if launched with a *-chim* flag), or when the user presses the "RTOS" key on the Onika control panel. In either event, the user is prompted for the name of the socket as well as the machine name of the RTOS (Figure 22).

When communication has been established, Onika assigns the new socket to the current configuration by pointing the CONFIG field element *enet* to the new SFD node. This socket is referred to as the *session socket* for the configuration. (Each configuration has the ability to have its own RTOS session, allowing the user to control more than one sub-


Connect to RTOS

Do you want to have an interactive session
with your real-time operating system? If so,
launch your RTOS on a machine, type 'network' at its prompt,
enter the address of the RTOS machine below,
and press the 'Connect to RTOS' button.

Name of RTOS: chimera

RTOS Internet address: grond.robots.cmu.edu

Figure 22: Connecting to the real-time operating system.

system.) Onika then asks Chimera to create another socket for the configuration, called the *signal socket*, to which it will attach. The session socket is used for Onika-initiated communications, whereas the signal socket is used for Chimera-initiated communication. The latter generally takes the form of error messages, warnings, and notifications from Chimera; follow up on messages received from the signal socket are done via the session socket.

Onika-initiated communications have the following format: Onika sends a message which asks for information or which issues a command. It then blocks on receiving a return reply. The return reply may be data which Onika requested, an acknowledgment that a command was issued, an error message, or a request for more data before completing a command. Onika checks the message type and reacts appropriately, by aborting its current activity (and informing the user) if the return message was an error indicator, issuing another data transmission if Chimera requested more data, or continuing its current activity if an acknowledgment or other form of conversation termination was sent. In any event, Onika always is the first to initiate a conversation on the session socket, and always receives the last reply.

Chimera-initiated communications always occur on the signal socket, and are always one-way. Onika continually polls this socket, waiting for notifications from Chimera (such as "task has completed successfully and turned itself off" or "the robot has lost power"). Further communication (if necessary) is thereafter performed using the session socket, freeing the signal socket to used for any further notifications. By routing Chimera-initiated warnings to a separate socket, currently ongoing communications on the session socket are not intermixed with out-of-context messages, and allow notifications to be stored separately when the session socket is otherwise busy or blocking. This also allows warning signals to be processed quickly, with little parsing necessary. This is especially important when applications are being executed, as discussed in Section 5.9.

5.8.3 Task-to-subsystem communication

End-users of programmed applications require a variety of types of feedback and input capabilities for quantitative testing and operation. For instance, a user may require a real-time data logger, or require that target trajectory endpoints be taken from a mouse-click location in a virtual reality display. As technology changes, capabilities such as these which are hard-coded into a visual programming interface may be rendered obsolete in a relatively short period of time. Subsequent support for additional types of I/O must be hard-coded within the visual programming interface; the interface designer must either program around existing constraints in his/her interface code (resulting in increasingly "hacked" update versions), or completely rewrite major portions of the interface (expensive in both money and man-hours).

To alleviate the problems associated with changing I/O requirements, it would seem essential to make I/O handling as modular as possible. Indeed, it would be best to move as much of the I/O handling as possible from the visual programming interface into other (external) subsystems. To date, this has not been possible with respect to real-time operating system frameworks. However, with the advent of port-based objects, we can now confine any I/O communication to the specific objects which require them. A port-based object can communicate directly with an external subsystem to receive input and send output, in a manner completely transparent to the interface. Since the port-based objects themselves are coded external to the interface, subsequent changes in I/O requirements for an object require changing only the code of the port-based object itself, leaving the interface unchanged (see Figure 23). This development significantly reduces the time required to support new types of displays and sensory inputs. Furthermore, the programming interface does not need to stop monitoring the job execution to deal with the nuances of the external subsystem. This is an important consideration, since an iconic user interface already devotes a significant amount of time to the maintenance of screen graphics. A similar mechanism is employed by hypermedia interfaces such as Mosaic, but has not, to our knowledge, ever been used with a real-time control interface before.

In our framework, we assume that any communication between a job and a specific exter-

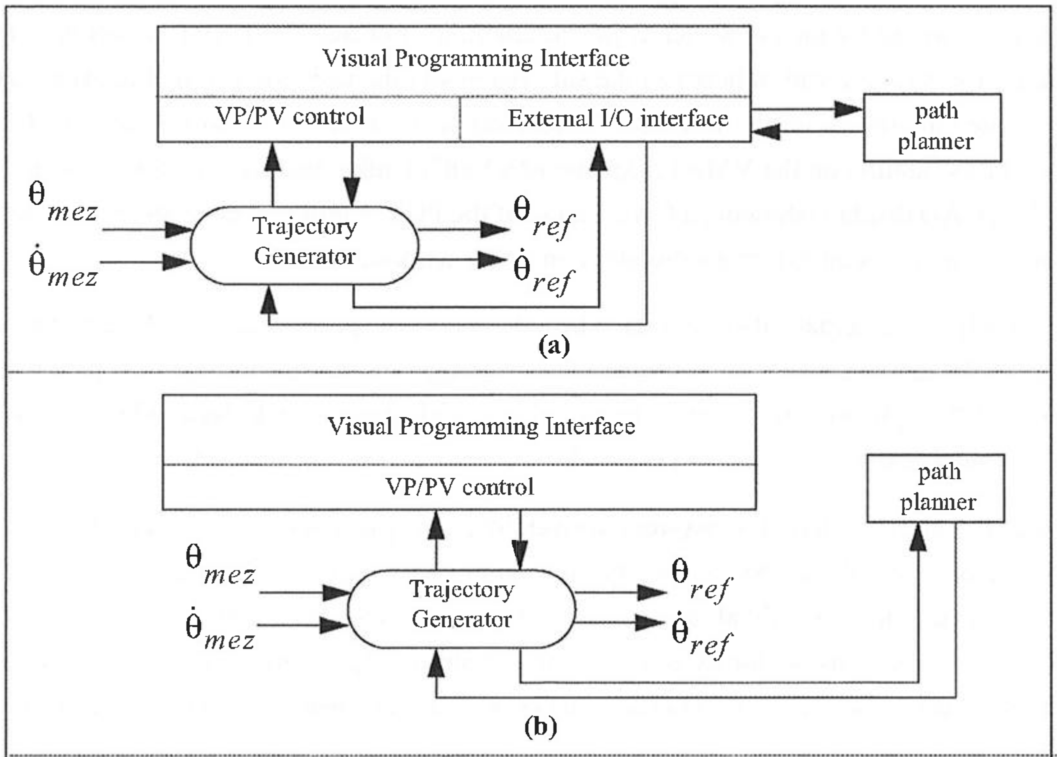


Figure 23: In (a), the retrieval and sending of I/O to and from the path planner is completely controlled by the visual programming interface. If the path planner I/O protocol is changed, then the visual programming interface itself must be changed. In (b), the path planner I/O is handled by the trajectory generator (a port-based object). If the path planner I/O is changed, then only the trajectory generator must be changed.

nal subsystem can be performed by one task within the job; either the task generates/uses the data which the external subsystem uses/generates, or can retrieve/pass the information from/to the table of system state variables. The task with which an external subsystem communicates is *synchronous*, rather than *periodic*; instead of operating at a fixed frequency, it sends a message via the Internet to the subsystem, and blocks on a response.[1] When the response is received, its data (if any) is analyzed and acted upon. The task then cycles, and the entire procedure is repeated until the task is deactivated (either manually,

1. One could also create a periodic task, designed to only output data, to simply send the current information without blocking. However, if the task is cycling faster than the external subsystem can process the data, then the socket will rapidly fill up, perhaps causing data loss, and at the very least ensuring that the external subsystem will be lagging the job, possibly rendering the data useless. For this reason, our convention is to always block on responses, to keep jobs and external subsystems synchronized.

through error, or by the controller at job completion). Because the ENET socket package is used for communication between the subsystem and the task, the external interface can be located anywhere on the Internet. A common setup in our laboratory is to have Chimera tasks running on the VME backplane of a Sun3, Onika running on a Sparc 10, and a PUMA CAD display (showing a CAD image of the PUMA either being controlled or simulated) running as an external subsystem on a SGI workstation.

Obviously, the external subsystem must be able to receive, process, and send the messages back to the task via sockets. This means that existing external subsystems (e.g. path planners, CAD displays) must be either modified or rebuilt from scratch. New subsystems can be designed with this mechanism in mind.

Using the above external subsystem convention, any particular job can interact with a number of external subsystems. If any programming interface (including Onika) were to attempt to do this, it would almost certainly be slowed to a crawl in its efforts to maintain all of the subsystems, seriously impairing its reliability. By eliminating the interface as a "middle-man," communication between tasks and displays becomes faster and more reliable.

5.8.4 Hypermedia communication

As mentioned in Section 5.2.1 and Section 5.4.1, Onika has the ability to retrieve tasks, jobs, and targets from remote file systems. Since the remote file system may not have a server which uses the ENET socket protocols, *ftp* is instead the method of communication used to retrieve files. Given a base directory on a remote file system, Onika creates a script file designed to retrieve all task, configuration, job, target, and application files from the various subdirectories. (Onika assumes a standard directory structure will be used; this is given in [22]). The script is then executed via a system command, and the files are copied to a local directory which is given the same name as the machine from which the files are being retrieved. Further system commands are then issued to automatically link any new modules into a Chimera executable, so that the modules can immediately be run by Onika and Chimera. Currently, the user is prompted for a login name and password before files are retrieved, but anonymous ftp could just as easily be supported.

The only drawback with this method for getting remote files is that they must be retrieved *a priori*; while Onika would have no problem loading in new libraries on the fly during the session, the Chimera RTOS is incapable of dynamically adding new modules to its libraries during a session. In anticipation of developments to Chimera which would support the dynamic addition of modules during a session, we have developed a true hypermedia interface for interacting with remote files during an Onika session. The functions we have created will allow the user to treat all files as "local" regardless of their location, and to be able to add new modules to a library during the session. To accomplish this, Onika will maintain an internal list of files aside their "true" locations, all transparently to the user. A remote file which is accessed by the user will be copied to a temporary file; subsequent references to the file will access the local "temporary" copy instead, as is done currently with other hypermedia packages such as Mosaic [51]. Changes made to the "local" copy can then be written back to the remote "real" copy, provided that the user has the proper permissions. Files which are already local will be accessed directly. The same high-level programming commands in Onika's source code, however, will be used to read from or write to any file regardless of whether it is "remote" or "local," with Onika figuring out exactly how this is to be done in the different cases. The code to support this "true" hypermedia interface is given in Appendix D. To our knowledge, the ability to access both local and remote resources in this manner has not hitherto been available in any other real-time control system interface.

5.9 Execution of Applications

Onika executes applications in a non-blocking fashion, allowing the user to fully monitor and control the outcome of applications. This is by using the signal sockets to poll for notifications, rather than blocking on the session sockets. Parallel applications may be executed on the RTOS, where each flow of the application will update its own configuration subsystem, which in turn will have its own pair of sockets. Furthermore, each task in an application may be conveniently aliased to another task, which means that (for example) any application can be run on any manipulator or on a simulator without any changes being made to the application. This functionality is unique to Onika. In this section, we describe the execution loop which allows this extremely flexible control of applications.

5.9.1 Simulation and redirection of applications

By clicking on the "start icon" of an application flow in a certain manner, Onika scans the application and displays all of the tasks which will be used by the flow during execution, including tasks to be used in nested subapplications. These tasks are displayed to the user in a pop-up window. Beside each task is a pop-up menu containing all tasks known to Onika which could conceivably replace that task (i.e. those which generate the same outputs). The value of the menu is originally set to be the task itself, but the user can select a different task to take its place. For instance, if the joint motion actions in the application flow all used a polynomial trajectory generator task, this task could be aliased instead to a cyclical trajectory generator task. When the application was subsequently run, the latter would be used whenever the former was encountered in the application flow. Each flow in the application has its own set of aliases, so that this "aliasing" can be localized.

The implications of this ability are very profound. Since, within our framework, actual control of the specific real-time control system hardware is localized to one task, and all other tasks are generic, we can alias this hardware task to another, and run the application on an entirely different system. For instance, if the application was created using actions designed for a PUMA robot, the user could alias the *puma* task to an *adept* task instead, and run the same application on an ADEPT robot. The generic nature of the other tasks within the applications means that they will automatically be updated with the information required to handle differing degrees-of-freedom (DOF) and Denavit-Hartenberg (D-H) parameters. The user can also alias the hardware task to a simulated version of the same task, thereby giving himself or herself the ability to simulate any application before actually running it on the real hardware. No changes in code need to be made; no porting of code from a real-time system to a simulator need be performed. Thus, errors incurred when porting code are eliminated, as is the (typically very large) amount of time necessary to actually port the code.

There are limitations on this ability, of course; applications cannot be aliased to run on system with gross physical dissimilarities (such as running a vision application on a system without a camera, or running a motion application on a stationary system. Such distinctions should be intuitive, but future developments on Onika (and its supporting RTOS) may include methods for determining the redirectability of applications *a priori*.

5.9.2 The execution loop

When a user requests that an application be executed on the RTOS system, Onika first checks the application to ensure that each flow in it contains no "holes," and that all programming structures are correct (for instance, ensuring that all loops have a beginning and an end icon). If any conditions are not met, the procedure executes gracefully, with the user being informed of the exact nature of the error.

Next, certain flags are initialized for each flow in the application. These include whether or not the flow is paused, in a loop, or in synchronization mode, and which default conditional path is to be taken for the flow if one is encountered. Additionally, pointers to the current sets of task aliases for each flow are stored in a global array. Finally, an *ONEX* node (the structure of which is shown in Code 10) is created for each flow. The *ONEX* node contains information about the application flow and about the configuration in which

```
typedef struct onex {
   struct onex *prev,*next;
   APPL *appl;           /* The application */
   ONIKON *icon;         /* The icon currently looked at */
   UI *ui; /* The input passed to all holes in that application */
   CONFIG *conf;
   int applnames; /* number of diff. applications recursed into */
   NAME_STRING applname[MAXRECAPPL];
   int confnames; /* number of diff. configurations recursed into */
   NAME_STRING confname[MAXRECCONF];
} ONEX;

extern ONEX *OnikaExecuting[MAXFLOW]; /* one list for each flow */
```

Code 10: The *ONEX* node, used for holding the current status of an executing application (and subapplications, via a linked list) within a flow.

changes are being made to traverse it. As each flow recurses into more deeply nested application, more *ONEX* nodes are added to the flow's linked list. These nodes are deleted as the relevant subapplications are completed. When the final *ONEX* node for a flow is deleted, then the Onika "knows" that the flow has completed its execution. When all flows have completed their execution, then the application terminates successfully, and the execution process ends.

Next, Onika clears all current configurations and assigns one to each flow in the application, creating new empty configurations as needed. If any of the configurations are not currently attached to the RTOS, Onika prompts the user for the information necessary to do so for each unattached configuration. (The assumption here is that the RTOS has been set up initially to accommodate the appropriate number of subsystems, which is a trivial thing to do.)

Once all flows have a configuration subsystem space in which to work, they are completely scanned (including all nested subapplications, if any) to determine the union of tasks required to completely execute each flow. These task sets are then spawned for each flow — if any tasks within a subsystem conflict with each other, the user is so informed, and the procedure exits gracefully. Tasks which are marked as having aliases (as mentioned in Section 5.9.1) have their counterpart tasks spawned in their place.

Traversal of each application flow is done by superset dynamic reconfiguration (described in Section 5.3.4). As each action[1] is finished, its trigger task flags Onika via the signal socket for its particular configuration. Onika analyzes the return value sent in this signal to determine whether looping should occur or which flow of a conditional application should

Page 117 of 198

be taken (as relevant). Onika then deactivates tasks no longer needed, and activates those tasks which are need by the next action (again, using aliased tasks as appropriate). For job-target tandems, any user input needed by trigger tasks in the job's configuration is automatically supplied by the target icon which follows the job icon (or application icon, as the case may be); if this input is invalid, the user is prompted to enter correct values. The user is also prompted for any other user input required which is not related to the trigger task, such as "power up the robot" and so forth.

At any time, the user can abort the application, pause any flow (after the flow's current action completes execution), or jump to the next action in a flow (convenient if the current action, for whatever reason,[1] has no trigger task to signal its completion, or if continued execution of a paused flow is desired.) When errors occur, Onika allows the user to either abort the application or to "clear and continue." If the latter is chosen, Onika clears the appropriate task (if possible), deactivates the trigger task (if any), and reactivates both tasks, giving the trigger task the same input which it was given during the previous attempt, thus repeating the step. If successful this time, the application proceeds apace.

If the current action in the sequence of the flow is marked with a breakpoint after it, then execution in that flow is paused after the action is completed. Execution may be resumed with a single mouse-click.

If, at any time, the next action is marked as a synchronization point, the flow is again paused, until a synchronization point having the same label is encountered in another flow, at which time the previous flow will be allowed to continue its execution. The flow may also be forced to continue by the user despite the existence of the synchronization tag if the user so desires.

If an action is marked as the test condition of a loop, its return signal is analyzed to determine if looping should occur. If so, the "current icon" pointer is repositioned to the other icon having the same label, and execution continues. If not, the action which follows the test action is taken, with execution continuing from there. Infinite loops may be executed by aborting the application or by forcing the flow to proceed to the next action in the sequence regardless of the value returned by the current action.

When the application has completed its execution, or whenever the application is aborted, all tasks in the various subsystems are killed, and the user is informed as to the nature of

1. We use the term "action" to refer to both complete jobs and job-target tandems in this section. When either type finishes, Onika updates the "current icon pointer" for the flow by either one or two icons, whichever is appropriate.

the outcome (successful or aborted).

5.10 Error Handling

There are three kinds of error handling within Onika. The first is *error prevention*, enforced by the syntax rules given in Section 5.3.3 and Section 5.5.3, as well as the pre-scanning given to potential task aliases and to applications prior to execution as discussed in Section 5.9. The second is *situational error handling*, such as dealing with tasks and applications which go into error, or tasks which are given incorrect user input, as discussed in Section 5.3.6 and Section 5.9.2 The final type of error handling, which has not hitherto been discussed, is communication error handling, which deals with errors in loading files and in ethernet communication. Onika uses the *err()* error package developed for Chimera 3.2 to handle these sorts of errors (the reader is directed to [57] for a full description of this package). In this section, we briefly discuss error handling of this type as is relevant to Onika solely for the sake of completion, while noting that the research and development for the detection of this class of error was actually pioneered in [57].

5.10.1 File errors

The files used by Onika have definite structures, all of which are given in Appendix A. There is a certain amount of flexibility built into Onika regarding these structures (for instance, Onika fully supports — and updates with defaults —files generated by previous versions of Onika which may lack fields found in the current file formats), but, in general, files must confirm to a rigid format. All Onika files are loaded in using the *cfig()* library routines developed for Chimera 3.2. Onika specifies the format of the file type, as well as the name of the file to be loaded, in the arguments to the *cfig()* call. If *cfig()* encounters an error while loading the file (for instance, corrupted data, missing lines, of other format violations), *cfig()* invokes an appropriate error handler in Onika, which will informs the user of the error and the steps which Onika will take to recover from it. For example, if a task's *.rmod* file is found to be in error when the task libraries are being loaded, the task will not be made available in the library, or if the user preference file is corrupt, Onika will assign "default" preferences for the session, which the user can modify if desired. The error handler procedure then gracefully aborts the current procedure, and allows subsequent actions (for instance, the continued loading of other tasks) to proceed as normal.

5.10.2 Network errors

All sockets within Onika are constantly polled to determine if they are active. If communication with the RTOS is lost at any time (even when Onika is blocking on a response from the RTOS), an error handler is invoked which informs the user that communication has been lost for one or more configuration subsystems. Any currently running applications affected by the loss of communications are marked and reported as "aborted," with their tasks deleted from their subsystems. If no application is running, but tasks were cycling within some configuration, these tasks are marked as "inactive." The socket information for all affected configurations is reset, and control is gracefully returned to the user. If the user subsequently reconnects a configuration to the RTOS, the Onika configuration will update itself with the current status of the tasks on the RTOS, resolving any differences by spawning new tasks as necessary, as discussed in Section 5.3.2.

5.11 Portability of the Onika Algorithms to Other Systems

Onika was designed with real-time control systems in mind; in particular, it is designed to work primarily with the Chimera real-time operating system. The algorithms presented and implemented within this research may be ported to other domains of computer science and engineering; however, within Onika, the following assumptions are made about the system for which it is a front end:

- *Lower level programming is state-variable based*: the current state of the system can be completely defined by the values is its state variables.
- *Upper level programming is goal-oriented*: Applications have one or more definitive endings, modified by loops and conditionals.
- *Any routine can be defined by (or decomposed into) a collection of common lower level routines*: The software at any level of the system is reusable, modular, and generic, and can be combined logically with other such software to create higher level routines.

It is possible that a system might not meet all of the above assumptions, and yet still be able to use exclusively either the upper level or lower level algorithms, since the syntax and grammar of either is separate (the execution routines within Onika provide the sole interface between the two).

5.12 Summary

In this chapter, we have discussed the implementation of tasks within Onika, and shown the rules by which they can be combined into configurations which perform some action. These actions can be "iconified" and included with icons representing targets in high-level dictionaries, from which goal-oriented applications can be developed. Task libraries, configurations, dictionaries, and applications may be retrieved from the local filesystem or remotely on the Internet; future versions of Onika will allow retrieval during a session (currently, the supporting RTOS only allows retrieval before the session begins). User preferences allow the user to customize his or her version of Onika, increasing the efficiency of the interface. Several novel communication systems have been developed to support Onika's communication with the RTOS, with other sites around the Internet, and between tasks and external subsystems such as real-time displays and path planners. These communication methods led to the development of a very fast execution control loop, which pre-scans all applications for errors, and allows for the redirection of any application to a different system or even to a simulator. Built-in error handling and prevention ensure that the Onika session will be "stable" despite any errors encountered in communication or in file loading.

6. User Testing

6.1 Introduction

In this chapter, we present the results from our user tests of Onika. These tests were performed in order to determine the efficiency of Onika as a programming environment. In particular, we wished to test whether programs in Onika could be created more quickly than programs created by conventional methods, regardless of the user's programming expertise. We also wished to discover what factors in Onika's presentation of information affected its efficiency. In the tests presented in this chapter, the subjects were divided into three groups, with each group using a slightly different version of Onika to determine which Onika characteristics most affected a subject's performance. The first group used a version in which the upper-level icons were presented graphically only. The second group used a version wherein the icons had an additional textual description centered beneath each icon. The final group used a version where the icons were display as featureless rectangles (i.e. no visual cues) with the textual descriptions centered beneath them.

The subjects were given a fifteen-minute tutorial in Onika's use and in basic robotic theory. After a warm-up test, the subjects were asked to program the robot to assemble a small structure using Onika within a time-limit of one hour. All users were able to correctly create the application, demonstrating that, using Onika and its support for iconic programming, end-users can successfully create and control wide-scope applications for robots without advanced knowledge of controls or computer programming. This is highly significant, since non-expert users would otherwise be unable to create code for real-time control systems using conventional methods.

As mentioned above, we tried to determine the importance of the various visual cues available in Onika by comparing the statistics compiled from the three different subject groups. Although we were unable to show 5% statistical significance with our limited subject pool, the comparison results were close enough (7%) that we feel that further testing is indicated to determine the importance of graphical and textual descriptions in Onika.

This chapter also introduces the first benchmarks for programming robotic applications from job primitives. Such benchmarks were never possible in the past, since the process of creating programs using textual methods tends to be too lengthy and monolithic.

6.2 Method

To determine Onika's efficiency as a programming interface, the subjects were asked to create a program using Onika that would cause the robot to assemble a simple four-piece structure. When assembled correctly, a light on the structure would light up, indicating success. To create the application, a minimum of 43 icons would be needed, incorporating 24 successive robotic actions. This simple task was chosen because it would be easy to compare the subjects' result times with the time required to generate the same application textually (using allegorical data in the latter case), and yet would involve using a large cross-section of Onika's abilities. This section details our methodology in running this experiment.

6.2.1 Subject demographics

For the testing, we chose subjects who met the following criteria:

- Windows-literate: all subjects had some experience using window-based interfaces, and could operate a mouse properly.
- College education: all subjects were completing or had previously received a Bachelor of Science degree.

Experience with computer programming and robotics varied within the subject group, and both men and women were included.

Twenty-four male and female subjects with varied experience in robotics and programming were tested in total, and were divided into three groups. The first group, subjects p1 through p9, used a version of Onika during the test in which individual icons in the job dictionary were differentiated from one another solely on their visual characteristics. The second group, subjects pt1 through pt10, used a version of Onika in which the icons were spaced farther apart, and were labeled with descriptions centered beneath them. The third group, subjects t1 through t5, used a version of Onika in which the icons were mere rectangles having textual descriptions centered beneath them, but having no other visual cues.

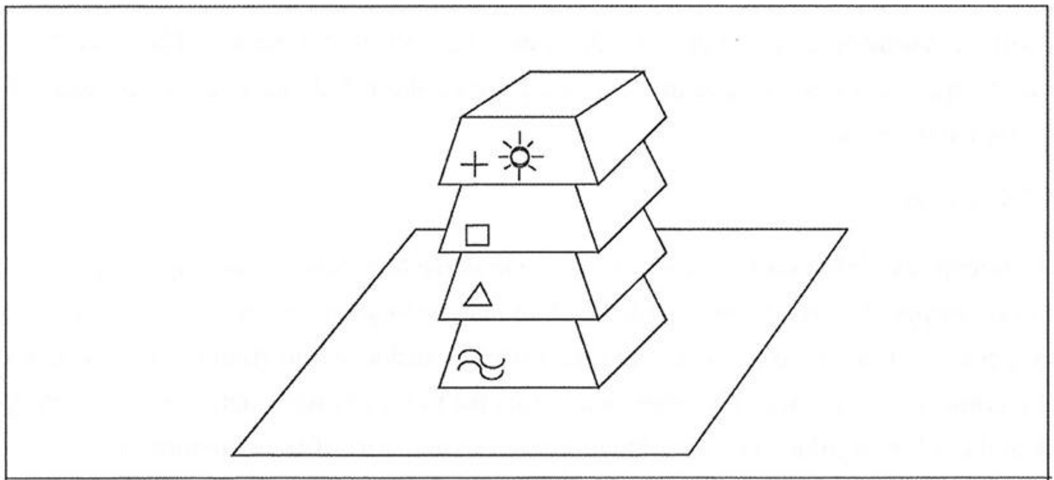


Figure 24: A depiction of the four boxes which the user programmed the robot to assemble in the user tests. Note that each box is labelled with a different symbol for the purposes of identification, and that the top box has a light in its center which illuminates when the structure is assembled correctly. In this figure, the boxes are shown stacked in the correct order.

6.2.2 Apparatus

6.2.2.1 Onika

All subjects participating in the experiment programmed the robot using Onika 1.38. Onika was run on a Sparc 10 operating as a peer on an NFS network. Connection between Onika and the supporting multi-thread real-time operating system, Chimera 3.2, was achieved using sockets which linked the interface to the OS across the NFS network. Chimera 3.2 was run on a peer Sun3 with a VME backplane supporting three RTPUs (real-time processing units). The arm controlled by Chimera 3.2 via the subject's programs assembled on Onika was a 6 DOF (degrees-of-freedom) PUMA 560 outfitted with a pneumatic two-fingered gripper. Onika's icon displaying procedure was modified as necessary to determine whether or not descriptions and visual cues were offered to the subject.

6.2.2.2 Structural hardware

The structure to be assembled consisted of four blue plastic boxes (electronics "hobbyist" boxes, 3 inches by 5 inches). Each box had a unique identifier painted on its "face" (wavy lines, triangle, square, cross, as depicted in Figure 24). Additionally, each box had metal contacts on the top and bottom, connected internally by wires. When placed on top of each other in the correct order, a complete circuit would be formed and an LED on the uppermost box would light up (the battery for which was stored in the bottommost box). Only the correct assembly of the boxes would cause the light to illuminate. The boxes were placed at specific locations within the workspace of the robot; these locations were the same for each subject.

6.2.2.3 Icons

The job icons available for the subject (*Cartesian move to x*, *joint move to q*, *open gripper*, and *close gripper*) were already available in the laboratory's user libraries, and had originally been assembled from reconfigurable software tasks within minutes using Onika's engineering-level interface. The approach points for the robot were determined by teleoperating the robot in joint space to a location above each part of the structure to be assembled, and recording the resulting position in a joint target icon. An Onika application was then created to move to each approach position and then switch to Cartesian teleoperation (in turn) so that we could easily move the hand to each target point and record the resulting location in a Cartesian target icon. Using both collections of target icons, it was then possible to quickly create a program to grab each part and move it above the assembly point of the structure, then switch to Cartesian teleoperation to lower each part to the structure to determine its "end point," which would also be stored in a Cartesian target icon. Using Onika, the creation of the approach and target point icons for the experiment took somewhat less than thirty minutes. The mnemonic pictures (9 colors available, 32 pixels on a side) were added afterwards, using a painting subroutine built into Onika.

6.2.2.4 Analysis Software and Criteria

To collect data, a background event logger was added to Onika's code. The event logger transparently generated a time-stamped entry for every action that the user performed using the mouse or keyboard (except moving or dragging of the mouse). The events were written to the log in a format which ensured that we would be able to completely reconstruct the user's session as required. The format included six data items: the absolute time (in seconds, from the system clock), the event (e.g., "select_icon," "insert_icon," etc.), the primary data structure affected by the event (e.g., a "move to x" icon), the primary memory pointer to the primary data structure (to pinpoint which instance of a primary data structure was affected), the secondary data structure affected (if any), its secondary memory pointer, and, finally, the window in which the event took place. As an example, the following data was generated when one subject selected icons in the dictionary and placed them into the application:

```
time        event          object      object's_ID  assoc. obj.  assoc. ID  window
770909683   try_new_appl   none        0            none         0          appl
770909685   new_appl       Untitled-2  3577432      none         0          appl
```

```
770909690   select_icon   opengripC   3496920    none        0         dict
770909692   insert_icon   opengripC   3652992    Start       3684304   appl
770909696   select_icon   jmoveC      3494072    none        0         dict
770909698   insert_icon   jmoveC      3762528    opengripC   3652992   appl
770909703   select_icon   box2_aj     3509360    none        0         dict
770909704   insert_icon   box2_aj     3782232    jmoveC      3762528   appl
```

The log file contained a listing of every OS (operating system) event which was generated by the user; however, the example data given above is clearly not in a readable format. Therefore, the raw data was parsed after any given test in two ways. First, the raw data was parsed to determine how long a subject spent performing any particular function in Onika. Since the number of events which a subject could generate was limited to creating new applications, selecting, placing, and deleting icons, and running their applications, this parsing program was able to provide a clear view of where most of a subject's time was spent. An example of this parsed statistical data from the same subject as in the previous example follows:

```
Time required to request new application:          2 seconds
Elapsed time from NEW to success:                  673 seconds
Total selection time:                              242 seconds
Number of meaningful selections[1]:                43
Average time per selection:                        5.62791 seconds
Total number of selections:                        43
Total number of insertions without selection[2]:   0
Time required for insertions without selection:    0
Total number of all insertions:                    43
Total insertion time (all):                        124 seconds
Average insertion time:                            2.88372 seconds
Number of deletions:                               0
Total check time:                                  59 seconds
Attempts to run the application:                   1
Applications which could not be run:               0
Applications which were aborted:                   0
Applications which were successfully completed:    1
Total time used by successful applications:        246 seconds
Average time used by successful applications:      246 seconds
```

By analyzing this statistical data, it became clear after a minimal number of tests that the average icon selection time was the major criterion in quantifying a subject's overall performance — insertion time for icons, for instance, was always within a half-second range. This statistical data was therefore quite useful in locating the major source of resource consumption; however, it was not at all useful for determining the reason *why* excessive time resources were being consumed. Without these reasons, effective enhancements to Onika's visual presentation could not be made. Therefore, a timeline was generated from

1. Selections which led to an insertion event, as opposed to selections which were followed immediately by another selection event.
2. An icon which is currently selected in the dictionary can be inserted multiple times in the application without needing to "reselect" the icon in between insertion events.

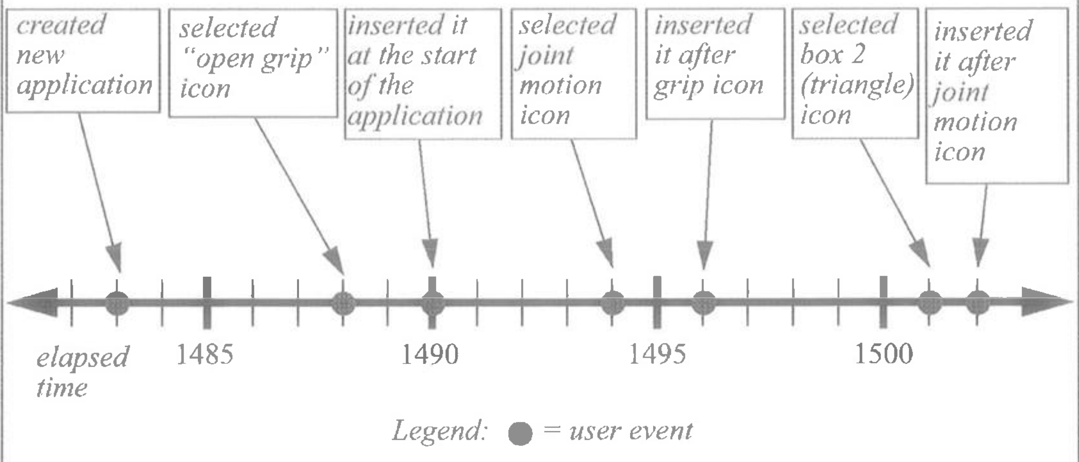

created new application
selected "open grip" icon
inserted it at the start of the application
selected joint motion icon
inserted it after grip icon
selected box 2 (triangle) icon
inserted it after joint motion icon
elapsed time
1485
1490
1495
1500
Legend: ● = user event

Figure 25: A graphical representation of a section of timeline data taken from a subject test.

the raw data, essentially rewriting it into a more legible format:

```
2 seconds later (at t=1483), the event new_appl occurred to item Untitled-
  2 (id = 3577432).
5 seconds later (at t=1488), the event select_icon occurred to item
  opengripC (id = 3496920).
2 seconds later (at t=1490), the event insert_icon occurred to item
  opengripC (id = 3652992) which followed/follows item Start (id =
  3684304).
4 seconds later (at t=1494), the event select_icon occurred to item
  jmoveC (id = 3494072).
2 seconds later (at t=1496), the event insert_icon occurred to item
  jmoveC (id = 3762528) which followed/follows item opengripC (id =
  3652992).
5 seconds later (at t=1501), the event select_icon occurred to item
  box2_aj (id = 3509360).
1 seconds later (at t=1502), the event insert_icon occurred to item
  box2_aj (id = 3782232) which followed/follows item jmoveC (id =
  3762528).
```

A visual depiction of this data is given in Figure 25. Using this timeline format, we could pinpoint (for instance) out-of-the-ordinary selection times for specific icons, allowing us to determine which icons were harder to find than others (i.e. too abstract). We were then able to effect changes to the Onika interface to provide more information to the user (such as adding textual descriptions, sorting icons, etc.).

6.2.3 Procedure

6.2.3.1 Preparation

Testing took place in the Advanced Manipulators Laboratory. Before each subject arrived, both Onika and Chimera were launched, and were connected to each other. The robot was calibrated and moved to the upright position with the gripper in the "closed" position. The visible Onika windows included the main control panel, the job dictionary, and the application workspace.

6.2.3.2 Administration of tutorial

Each subject first filled out a demographics sheet which indicated their relative expertise in the areas of programming and robotics, as well as their level of education. Next, the subject was given a 15-minute oral and demonstrative tutorial which included:

- what the finished structure would look like if assembled correctly (using a wooden mock-up of the parts of the structure)
- how to create a new application within Onika
- what each icon would do when encountered within an Onika application
- the reason for using joint-space approach points when causing a robot to move, and the difference between them and Cartesian-space target points
- how to place icons from the job dictionary into the application workspace
- how to use the scroll bar to work with portions of the application which might have moved off-window

A checklist was used to ensure that each subject was given the same tutorial and the same set of instructions. Additionally, each subject was given a "help" sheet which reiterated (textually) how to place icons from the dictionary into the application, and which also showed a picture of the finished structure (the sketch in Figure 24 was used for this purpose).

6.2.3.3 Warm-up

Next, the subject was shown how to make an application which would cause the robot to pick up the first part of the structure. After this application was run, and the robot was reset, the application was removed, and the subject was asked to re-create the application. If a subject's warm-up program failed during execution, the subject was told why (e.g., "the program has failed, because the robot failed to grab the box before picking it up"), the robot and structure parts were reset, and the subject was asked to repair the program. The subject was allowed three attempts to correctly assemble the warm-up program.

6.2.3.4 Assembly of structure

Having succeeded in creating the warm-up application, that application was deleted, and the robot and structure parts reset. The subject was then told to create an application to assemble the entire structure within a one-hour time limit. If execution of the application failed, the subject was told why (as in the warm-up), the robot and pieces of the structure would be reset, and the subject was then free to revise the application within the one-hour constraint.

6.2.4 Results

All subjects in the first group (p1 through p9) succeeded in recreating the warm-up application. Additionally, with one exception, all subjects were able to successfully assemble the structure using their Onika program on the first attempt. The exception, subject p8, left out an icon pair. Subject p8 was able to locate and correct his error in a minimal amount of time (165 seconds), and subsequently succeeded on the next attempt to run his application.

Originally, we had not planned on varying Onika's functionality during the user tests; all subjects would be tested using the laboratory version of Onika, in which icons did not have textual descriptions. However, by analyzing the data generated from the tests of subjects p1 through p9, we had been able to determine that the majority of a subject's time was devoted to selecting an icon. Based on this information, we added textual descriptions centered beneath each icon in the job dictionary, and we spaced the icons farther apart, all to determine whether the selection time could be reduced. The resulting version of Onika was tested on subjects pt1 through pt10.

The performance of subjects pt1 through pt10 was well within the range of those of the first group (subject pt4 required three warm-up attempts, and subject pt2 forgot an icon pair, as subject p8 had — the correction was made in 43 seconds). A major exception was subject pt9, who failed the warm-up three times and was therefore eliminated from taking the full test. Despite the fact that, by our initial testing criteria, the subject's full test results could not be used when tabulating results, we explicitly walked the subject through the warm-up a fourth time, after which the subject was able to complete the warm-up. We then allowed the subject to take the full test, which was completed successfully on the first attempt, albeit slowly when compared to the others in the same (i.e. icons with descriptions) test group (average time of 13.2 seconds per icon selected, as opposed to a group average time of 6.1 seconds per selection).

When comparing the group which used icons only to the group which used icons that had textual descriptions, we found that the mean performance increase of the latter group with respect to selecting icons for placement into the application was 22% (6.1 seconds versus 7.8 seconds).Although this difference did not prove to be significant, as discussed later in Section 6.2.5, the probability of significance was close enough that further testing is indicated.

The completion times for the subjects using the icons with textual descriptions ranged from 257 to 868 seconds, whereas for the subjects using just the icons, the completion times ranged from 338 to 1038 seconds. The tabulated results for the "no descriptions" group and the "descriptions" group are given in Table 1 and Table 2, respectively. We then determined to test how much on an asset the visual cues were in and of themselves. To do this, we created a version of Onika in which all icons were presented as blank rectangles having no pictures, edge colors, or edge shapes, but only textual descriptions.

..

Table 1: Performances when no descriptions were given

subject number	avg. selection time (sec)	number of icons deleted	number of attempts to run program	elapsed time (not including run times) (sec)
p1	8.55	4	1	655
p2	16.0652	3	1	1038
p3	3.89	0	1	376
p4	3.96	1	1	338
p5	9.33	0	1	730
p6	9.64	0	1	774
p7	7.13	0	2	641
p8	5.83	0	1	546
p9	6.19	0	1	428

Table 2: Performance when descriptions were given

subject number	avg. selection time (sec)	number of icons deleted	number of attempts to run program	elapsed time (not including run times) (sec)
pt1	5.894	2	1	521

Table 2: Performance when descriptions were given

subject number	avg. selection time (sec)	number of icons deleted	number of attempts to run program	elapsed time (not including run times) (sec)
pt2	6.067	0	2	659
pt3	5.628	0	1	427
pt4	7.574	0	1	603
pt5	6.720	0	1	568
pt6	4.733	0	1	379
pt7[a]	1.978	0	1	257
pt8	10.130	0	1	868
pt10	5.979	1	1	413

a. Note that icons *are* selectable even when they haven't all been "drawn" yet during a refresh. This accounts for subject pt7's seemingly-impossible low selection time (all other subjects waited until all icons were redrawn before trying to select one).

Table 3: Performance when no graphics were given

subject number	avg. selection time (sec)	adjusted selection time (sec)[a]	number of icons deleted	number of attempts to run program	elapsed time (not including run times) (sec)
t1	18.403	20.653	8	2	1375
t2	6.250	8.500	0	1	386
t3	6.385	8.635	0	1	441
t4	8.143	10.393	1	2	684
t5	6.750	9.000	5	1	542

a. Provided for comparison with Tables 1 and 2, since screen redraws for this version of Onika were less by 2.25 seconds per refresh event. The screen refreshes after icons are selected, dragged, etc., and subjects typically waited for the screen to refresh before performing any action (see footnote for Table 2). Hence, a 2.25 second penalty has been added to produce an adjusted selection time suitable for comparisons with other versions of Onika. (Note that, irrespective of refreshing delays, the performance of these subjects is clearly more error-prone than that the other groups.) The quicker refresh times should not be considered an advantage of using a "text-only" presentation, since, on a faster graphics workstation, the screen refresh rates would be negligible for Onika in any event.

The modified Onika, having no detailed graphics to draw, was freed from the 2.25 seconds (on average) of screen refreshing time always encountered when an icon was selected, giving these new subjects an advantage in time over the previous subjects. Nevertheless, the new subjects performed more slowly than the previous subjects on average, and they encountered more problems. In particular, the average number of incorrect icons placed into an application, and then subsequently deleted, per subject tested rose from 0.33 icons/subject for the pictures/descriptions case and 0.89 for the pictures/no descriptions case to 1.8 icons/subject, a highly significant change for the worse. The average selection time rose to 9.18 seconds even with the 2.25 second advantage, giving the "best case" scenario of both pictures and descriptions a 5.3 second advantage over the "text only" case -- a 47% time savings, not including the time penalty accrued by placing incorrect icons which were subsequently deleted. Again, although the difference between the different groups was subsequently not shown to be of statistical importance (Section 6.2.5), the statistical difference was found to be small enough that further testing was indicated. The tabulated results for this text-only group are shown in Table 3.

The two least proficient subjects (t1 and t4) both had to run their applications twice, with the least proficient subject (t1) succeeding only marginally on the second attempt (several icons pairs representing robot approach points were omitted; however, these points were non-critical, unbeknownst to the subject). The number of deletions and the selection times also reflected the need for proficiency in computers when using this version of the interface, as the most errors and the worst times directly reflected the user's experience with programming (a sophomore CS student performed best, two freshmen with high school programming experience performed less well, and the two non-programming subjects performed significantly worse than the others). This is unlike the cases where pictures were provided for each icon, in which proficiency seemed to matter very little to performance. (Of the nine subjects in the graphics-only group, four were "proficient" in programming, whereas in the graphics/text group, five of the eight were "proficient." The resulting scores could not be correlated to experience.)

6.2.5 Significance of the statistical data

In order to determine the significance of the data collected from the user test, an "analysis of variance" test (ANOVA) was performed on the data, with the outliers (subjects p2 and t1) removed. It was determined from our limited data set that the probability of significance of the data (using a sum-of-squares method) was 6.99%, and is therefore not significant with respect to a 5% probability criterion. Nevertheless, the results are close enough to being significant that further testing with a larger and more balanced subject pool may very well verify the perceived differences in the various versions of Onika; further testing is indicated. The results of a Tukey's Studentized Range (HSD) test on the data are shown in Table 4.: The versions of Onika in which graphics are used do not show a significant

Table 4: Subject group comparisons (HSD test)[a]

Group comparison	Simultaneous Lower Confidence Limit	Difference Between Means	Simultaneous Upper Confidence Limit
Graphics and text to graphics-only	-3.2796	-0.7369	1.8058
Graphics and text to text-only	-6.1984	-3.0539	0.0907
Graphics-only to text-only	-5.5214	-2.3170	0.8874

a. Adjusted text-only scores, as defined previously, were used in this testing.

difference, but the comparison of either to the text-only group is much more interesting, particularly graphics and text vs. text-only case, where the confidence range barely straddles the zero mark. Again, further testing is indicated.

In the next section, we investigate the subjects' performances further as we discuss the test results within the contexts of cognitive style and performance motivation.

6.3 Discussion

The main conclusion we derived from the test results was that Onika, in combination with its support for iconic programming, allowed users of all levels of expertise to program and control real-time control systems, even in cases where the subject clearly would have been unable (due to lack of training) to do the same usual conventional textual methods. In this section, we discuss reasons why using Onika is significantly less-resource intensive than using more conventional real-time programming options.

6.3.1 Usability of Onika by non-programmers

As mentioned previously, the control and programming of real-time control systems is by-and-large unachievable using conventional programming methods. The subject testing in this chapter, however, shows that even users who have never used a computer for other than word processing can successfully create programs for a robot. Furthermore, since new upper level icons can always be created from lower level icons, the scope of the end-users' programs is never limited, unlike interfaces in which the language elements are hard-coded.

It is easy to argue that the visual presentation of the icons assists in creation of the code. Subject pt4 commented that he narrowed down the field of potential icons using the color/ shape information, and used the textual descriptions to make certain that he picked the correct icon of the set. Subjects p1, p2, and p3, who used the icons without descriptions, suggested that descriptions should be included to make it easier to find icons. The importance of these results and comments cannot be understated, as many human-machine interfaces offer only one or the other sort of cue, which may therefore provide insufficient guidance to the user and may minimize system effectiveness.

6.3.2 Reduced set-up time

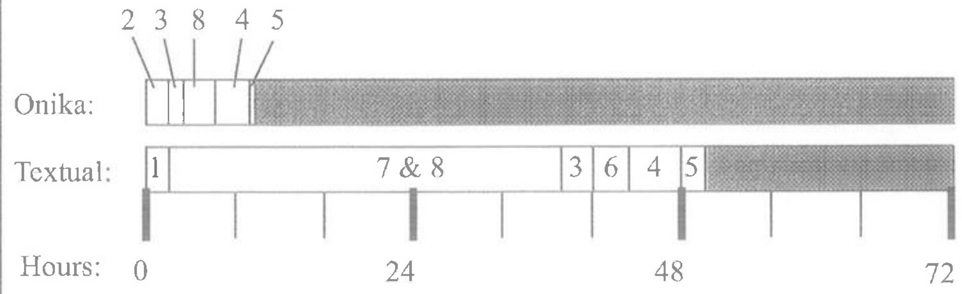
Figure 26 shows a comparison of the time required to build an Onika real-time pick-and-place application from scratch as opposed to building the same application using conventional methods.[1] The time required to create and assemble all of the software for the subjects' tests is compared to the creation of similar code for VAL II from scratch. As can be seen, there is a significant difference in the time required to generate end-user code, since Onika already supports real-time code, and Onika code need not be rewritten in order to be simulated. The stratified architecture which was created for Onika allows for a smooth decomposition of tasks in a manner which minimizes the amount of code which must be created for an application.

Besides basic issues such as the presentation of icons and information as well as the modularity of the architecture, there are other factors involved which may have played a part in determining a subject's results. In this following sections, we investigate two of these: cognitive style differences and performance motivation.

6.3.3 Effects of cognitive style upon performance

All subjects (subject pt9 excepted) succeeded in learning to create a robotics application

1. Anecdotal evidence from several robotics labs was collected to produce a time-line for conventional programming; as mentioned previously, no quantitative benchmarks are available for textual programming. The assumption in both cases is that device drivers are available to use in programming. The graphic would be even more skewed in Onika's favor if we account for the fact that the generic software already available in Onika libraries is sufficient for all robotic motions, without needing to rewrite it for different robots. Furthermore, tutorial times for the system are omitted; these would be negligible in Onika compared with the days of training required for the textual system.



Legend
1 - Create high-level simulation
2 - Create new modules
3 - Create configurations/job primitives
4 - Create/get target information
5 - Create application
6 - Port between simulator and physical system
7 - Create real-time wrapper/controller for new primitives
8 - Debug

Figure 26: A comparison example of creating an Onika real-time pick-and-place application versus creating a similar application using traditional methods. No simulation port is required for Onika, since physical code can be simulated without changes. Furthermore, unlike the textual case, the Onika code can, after completion, also be executed on another manipulator without changes. As noted previously, training times are omitted from this figure; this would be negligible in Onika compared to the days of training required for textual systems.

within a time frame typically unachievable by even expert programmers when using traditional textual languages. We speculate that subject pt9 was not able to adopt a strategy consistent with his *cognitive style*. The cognitive style of a person is either *field-independent* or *field-dependent*, depending on the methods he or she typically uses to solve problems[7][65]. Field-independent persons generally use a strategy based on analysis, hypothesis, and experimentation, all very important elements in a test environment where the test administrators were not permitted to volunteer information or advice to the subject beyond the initial reading of the test script. Persons with a field-dependent cognitive style tend to use strategies which rely on external referents, which were not available in this test except in a very limited fashion. The tutorial which we gave the subjects prior to the test (and, indeed, Onika itself) are well-suited to a field-independent cognitive style. The "help" sheet given to the subjects, for instance, was deliberately vague, containing only a picture of the completely-assembled structure and a reminder of how to drag icons from the library to the application. Subject pt9 consistently tried to use the test administrator as an external referent, and made it clear that he was trying to remember what the test administrator had done while demonstrating Onika before the warm-up, all of which are characteristic of a person having a field-dependent cognitive style. This behavior was not observed in any of the other test subjects.

To appeal to the field-dependent types, the help sheet should have been more detailed, with perhaps the example of picking up the box included on it for future referencing. This idea of creating different training environments for persons of differing cognitive style is supported in [31]. A field-dependent person who could not switch to a field-independent cognitive style would indeed be left with no other alternative but to try to reconstruct from memory what we had demonstrated earlier, or wait for further cues. The use of computers and robots can certainly be intimidating and stressful, making it difficult to adopt a more suitable cognitive style strategy[7]. The subject might also have been "fixed" rather than "mobile" with regard to cognitive style; proper training on Onika would likely have helped the subject to become able to access the qualities of the foreign cognitive style[65]. We do not mean to imply by this that cognitive style is associated with cognitive ability; previous research such as in [31] shows that the two seem to be independent.

6.3.4 Performance motivation

During the recruitment process for the testing, many potential subjects of varying proficiency indicated that they were highly motivated to participate in the test, and that the monetary compensation was not a factor in their volunteering to run the robot. After the test was given, however, the feedback from the subjects varied greatly according to group. The "non-picture" group of subjects was highly vocal in their criticism of the interface, with most of the criticism being negative. Four of the five subjects complained that the syntax was difficult to understand, and it was suggested that more visual cues be presented to the user. The more-proficient subjects in that group still called Onika "fun," but the non-proficient subjects were discouraged, and stated plainly that the interface was "not fun." All suggested adding features which, unknown to them, were already incorporated into the "normal" version of Onika. This is in direct contrast to the comments from the subjects in the other groups; subjects pt1, pt8, pt9, p5, p6, and p9 indicated after the test was complete that the interface was "fun" to use, regardless of their proficiency, with no negative comments received at all.

The idea of users having "fun" while performing complex tasks on a computer is an aspect of programming which we feel tends to be overlooked. A user who enjoys programming is invariably more productive than a user who looks upon it as a chore. Ten subjects had never written a program before, and thus were able to achieve something otherwise unavailable to them, i.e., write a program to directly control a robot. To these persons, Onika (and the test) might have appeared as a sort of game (which might have made it easier for certain subjects to adopt an appropriate cognitive strategy). The Onika test, for them, would contain many of the features found in the most popular games [43]. These include:

- an explicit goal (assembling the structure)
- a scoring system (implicit in the test itself)
- audio and visual feedback (built into Onika, plus the satisfaction of seeing and hearing the robot move)
- reliance on speed (the arbitrary one-hour time limit imposed on the test subjects).

Malone *et al* point out in [43] that explicit goals produce more enjoyment if the goal is interesting; all of our subjects responded to our initial request for testers that they were highly interested in robots and computers, even if they lacked previous experience with them. Several subjects offered to take the test for free. The subjects were highly motivated to achieve the goal of using a robot correctly, and the task was clearly intrinsically motivating in the following areas, defined in [43]:

- *Challenge*: "Can I make the robot work?"
- *Curiosity*: "I wonder how they programs robots."
- *Control*: "I have power over this hitherto inaccessible device!"
- *Fantasy* (both the fantasy of using a robot, and having the material assembled in a storyboard fashion): "I'm writing a story for a real robot to read, using pictures!"

For the users in the "text-only" group, this motivation clearly waned (especially in the area of "fantasy") as the subjects felt they were forced to use an interface which they could see was inherently flawed (based upon their suggestions for possible changes). We feel that it is clear, therefore, that attention to performance motivation factors is essentially to providing a useful human machine interface.

6.4 Summary

Based on the results of our user testing, we have shown that Onika is an effective tool for creating goal-oriented programs for robots, even with a minimal amount of training. In the past, the creation of real-time goal-oriented programs for robots has been measured in days, weeks, and even months, and has always been limited to specialists in the robotics and computer programming fields. For example, the real-time textual program required to assemble the structure used in this experiment took one roboticist in our laboratory several weeks to create and debug, and the code thus generated had a length of eighteen pages on paper. Programming of robotics by non-specialists, where even possible, has been extremely limited in scope due to the amount of training required. Nevertheless, in our subject tests, non-specialists were able to create an assembly application in less than 18 minutes. The supporting architecture for Onika assists in reducing set-up time, while the graphical interface is easy to understand and imposes little training time. Future work in this area should be directed towards using testing to determine how to make the presentation of higher-level programming structures (such as loops, macroing, synchronicity, and conditionals) more intuitive to non-specialist operators of robotic manipulators, as well as towards determining whether the visual presentation of icons in Onika truly varies according to whether graphical and/or textual information is presented.

7. Summary, Contributions, and Future Work

In this dissertation, we have presented a *visual programming environment* called Onika, which is designed to both significantly decrease the time and resources required to program and control complex real-time control systems. As a result of the research and development of the system presented in this dissertation, we have made the following significant contributions to the fields of real-time control systems and human-machine interfaces:

- A significant decrease in the time required to program real-time control systems has been achieved, with the time required to create high level applications measured in minutes and seconds rather than in hours and days, as shown by event-level user testing.
- Wide-scope programming of real-time control systems, such as robots, has been made accessible to a group of persons who lack the prerequisite training to perform such operations using traditional programming methods. This programming is done with the context of a well-defined system architecture.
- Resources and code for real-time systems can be shared transparently across a network, allowing a much faster transfer of technology. Approximately 100 generic software modules are already accessible by Onika users, covering a wide range of tasks.

Onika is already in daily use at Carnegie Mellon University, as well as at other sites around the continent, and has been instrumental in getting new systems up and running in hours rather than weeks.

Although the work presented in this thesis is a strong, positive step towards better programming and control of real-time systems, there are still many issues related to visual programming environments which need to be addressed in future research. These issues include, but are not limited to, the following:

- Onika currently recognizes at most one trigger task per job, thereby allowing each job to have only one target assigned to it. A mechanism needs to be devised which will allow a job to have more than one target icon modifying it. This will have the added benefit of making nested applications more useful, as more than one "argument" will be able to be passed to them (i.e. the programmer could leave more than one type of hole in an application, and "iconify" it for other applications to use, passing more than one target icon to it). To achieve this, the visual presentation of the upper level of Onika will need to be substantially modified.

- Tasks which require external subsystems currently rely on the user to start these subsystems *a priori*. A mechanism should be devised by which a hyperlink to an external subsystem could be assigned to any given task, so that when the task is activated, the subsystem is automatically spawned with the proper arguments.

- As mentioned in Section 5.8.4, Onika only retrieves software from across the Internet at start-up, due to limitations in the RTOS which prevent the dynamic loading of object code. When this limitation is overcome, the interface should be modified to present a true on-the-fly mechanism for interacting with software and hardware on the Internet. Code for this mechanism has already been developed (and is listed in Appendix D), but has not yet been included into the system.

- Currently, environmental variables are used to specify software locations. As more and more modules become available via networks, an interactive hypermedia browser will be required to determine which libraries should be opened.

- Onika, while object-oriented, is nevertheless written in C, and uses large, inefficient standard library routines to present graphics to the user. A logical development would be to rewrite Onika to use an object-oriented language (such as C++) and to custom-design fast graphics routines to reduce refresh rates. Currently, screen refreshes take 0.125 seconds per icon on the screen in the upper level; with 50 or more icons in a dictionary, the refresh time is longer than the time it takes the typical user to actually select an icon.

- A mechanism needs to be devised by which Onika can automatically launch the RTOS, even if the latter is on a different filesystem. Currently, the Chimera RTOS does not support remote launches, but this limitation will be addressed in future research, and Onika will need to support it when available.

- Automatic transfer of current SVAR values (such as the current joint position) to new target icons would significantly enhance performance. This mechanism would have to account for differences between the target syntax and the SVAR (for instance, joint targets require a trajectory duration as well as an endpoint).

- Onika currently assumes that the size of a given SVAR structure is constant. With the introduction of reconfigurable modular manipulators, this is no longer the case; as change in the value of NDOF will requiring resizing other arrays such as DH and measured joint positions. Task modules can already be written to compensate for this, but target icons currently cannot (for instance, a joint target as defined in our own preferences has exactly six joint values plus a duration value, and hence is unusable for any manipulator with more than six joints). Further investigation is required to produce target definitions which can automatically resize their arrays based on the current control system in use.

- Certain tasks are often used together (for instance, in our laboratory, robot tasks are always used with a gravity compensation task and a differentiator). These tasks could be grouped together in a "supermodule" which the user could manipulate as a whole. Supermodules would include rules as to the order of spawning, order of activation, etc. If and when implemented, Onika will need a way to group the component modules together, add the various interaction rules, and save, recall, and "expand" the supermodules as necessary. Ideally, this research and development would be performed in parallel with RTOS developments supporting supermodules at a lower level.

- Conditional support in Onika is fully functional, but only in a "proof-of-concept" form. More research and development needs to be done to devise a method to present conditionals intuitively in a way which does not waste large amounts of screen space (of which traditionally-touted flow charts are particularly guilty).

- Through user testing, we have discovered that minimal changes in the features of icons affect performance. More user testing needs to be performed to pinpoint the features of icons within a visual programming environment which most affect the usability of the interface.

- Currently, interactive debugging is limited to checking for memory corruption by tasks, simulation of applications and jobs, clearing tasks in error, display of feedback from the RTOS, and stepping through applications using breakpoints. A mechanism for tracking down specific errors in module code (and repairing them from within Onika) would greatly enhance Onika's use as a debugging tool.

- Real-time robotics has very specific state variables, which are well-characterized after years of testing. Onika's interfaces are therefore geared towards a state variable-oriented system. Nevertheless, both the lower level and upper level algorithms may be useful in other (non-real time) domains. Further research into the applicability of Onika's interfaces to other domains is indicated.

The introduction of reusable software for real-time control systems can significantly increase the usability of such systems and the rate of technology transfer, but only if the control and programming of such is made accessible to a wider cross-section of the population. A visual programming environment designed to ease the chore of creating code for these systems, such as the one presented in this dissertation, will not only conserve resources devoted to technology transfer, but will reduce the costs associated with training and the research and development of new applications in academia, manufacturing, and industry.

Appendix A: File Formats

This appendix lists out the various file formats used by Onika. All file formats are compatible with Chimera 3.2 library routines.

Note that all files (except the task parameter file) contain a *VERSION* field. If this field is not found within a file, then Onika will assume that the file was created by a version of Onika that was less than 1.3, and react accordingly (updating the file when re-saved, for instance).

In the following file formats, italics indicate that the italicized phrase is replaced by appropriate values within the file. Non-italicized words appear exactly as they do within the file.

A.1 Task Parameter (*.rmod*) file

This is the format for the task parameter file (*base_directory/module/xxx.rmod*):

```
##### Comment line

MODULE          name of task's module
DESC            description of task.
SVARALIAS       coded_name_1 = svar_1
SVARALIAS       coded_name_2 = svar_2
(...)
INCONST         list of coded INCONSTS
OUTCONST        list of coded OUTCONSTS
INVAR           list of coded INVARS
OUTVAR          list of coded OUTVARS
TASKTYPE        periodic,synchronous, or aperiodic

FREQ            frequency
    (or)
PERIOD          period

ONIKA
OBJECT          type of job this task implies

LOCAL
CODED_PARAMETER  parameter values
```

```
EOF
```

# denotes a comment. Blank lines are treated as comment lines. Combinations of tabs and spaces are treated as one space.

MODULE contains the name of the object file (minus the *.o* extension), and is used by Onika 1.3 to inform the RTOS as to which objects should be loaded. It must be the first non-commented item in the parameter file. The name of the object need not be the same as the name of the parameter file; for instance, the first item of a different parameter file, *her_task.rmod* might have *MODULE a_task* also, thus allowing the user to easily set up different tasks which use the same module by setting up different parameter files for that module. The module name should never be changed in this file.

DESC is largely irrelevant to Onika 1.3, and is only used to inform the user as to what the module in question does. Nevertheless, this item must be the second item in the parameter file.

SVARALIAS is an optional parameter. It allows the user to assign new names to internal variables. For example, the code for the module *a_task.o* might use the variables *JOINTS* to describe degrees-of-freedom and *DH* to describe Denavit-Hartenberg parameters. However, we might prefer to refer to these variables as *NDOF* and *DHPARAMS* instead. *SVARALIAS* allows us to do this. Subsequent references to these variables in this parameter file use the new definition rather than the old. *SVARALIAS* is especially useful when porting tasks from other worksites, where their standard variable names are different than yours. It's important to change names to match you own standards, so that Onika 1.3 will know when two variables refer to the same data. There may be several *SVARALIAS* lines in a parameter file, or there may be none. If there are any, they must follow the *DESC* parameter line.

INCONST, OUTCONST, INVAR, and **OUTVAR**: These refer to the inputs and outputs to and from the module.. They must respectively follow any *SVARALIAS* parameter lines; if there are no *SVARALIAS* parameter lines, then they must follow the *DESC* parameter line instead. If any one of them have no values, then the word "none" must follow it. These values represented the coded names of the variables; if there are *SVARALIAS* entries which refer to them, then Onika will automatically replace the affected variables with the aliased name when displaying and controlling the task.

TASKTYPE is one of the following: periodic, synchronous, or aperiodic. This line is *not* optional, and it follows *OUTVAR* parameter line.

FREQ or **PERIOD** is the default frequency or period of the task. Exactly one of these lines is included if and only if a task is periodic or synchronous. The use of one line over the other is based on user preference. When included, this line follows the *TASKTYPE* parameter line.

ONIKA signals that the next block of data is useful only to Onika 1.3. It need not exist in the *.rmod* file, but if it does, it follows *FREQ/PERIOD* parameter line (or the *TASKTYPE* parameter line, if FREQ/PERIOD is omitted). *ONIKA* is always followed by the *OBJECT* parameter.

OBJECT specifies the type of job this task defines. That is, if this task is part of a configuration, then that configuration should be considered as a job of the type listed on this line. Generally, if a task requires user input, this means that the job it is in is should be considered an action requiring an certain type of object. This type of action is listed in the *OBJECT* parameter (e.g. *Joint job*). Onika 1.3 uses this field to determine an upper level job's type, and to determine what kind of icons may follow it at the upper level. If a task requires user input, but this is not mentioned in the *.rmod* file, then, instead of Onika 1.3 getting the data from the icon following the job which includes this task, the user will be prompted for the data. *OBJECT* always follows *ONIKA,* and precedes *LOCAL* if the latter exists.

LOCAL signals that the final block of data is useful only to this task's module. *LOCAL* follows *ONIKA* and *OBJECT* if they exist; otherwise, it follows *FREQ* or *PERIOD* (or TASKTYPE if they are omitted). Following *LOCAL* come lines of data for the task's module. These vary from task to task. **DEVICEFILE** is an example of module data. The use of *LOCAL* allows the user to easily change the gains, devices, etc., associated with a module without having to recompile the module.

EOF must be the last line in the parameters file.

A.2 Task Configuration (*.conf*) file

Unlike the *.rmod* file, the *.conf* file is primarily used only by Onika itself. This is the format for the task configuration file (*base_directory/conf/xxx.conf*):

```
VERSION        version number of Onika in which this file was saved

RMODFILE       task parameter file of task_1
COORDS         x y
INLEFT         yes or no
FREQ           frequency
RTPU           processor

RMODFILE       task parameter file of task_2
COORDS         x y
INLEFT         yes or no
FREQ           frequency
RTPU           processor

(...)

LOCUS          svar_name_1
COORDS         x y
CONNS          conns
NUMINCONST     number_of_inconsts
NUMOUTCONST    number_of_outconsts
NUMINVAR       number_of_invars
NUMOUTVAR      number_of_outvarss
NUMDOUBLEIN    number_of_double_inputs
SHOW           yes or no

LOCUS          svar_name_2
COORDS         x y
CONNS          conns
NUMINCONST     number_of_inconsts
NUMOUTCONST    number_of_outconsts
NUMINVAR       number_of_invars
NUMOUTVAR      number_of_outvars
```

```
NUMDOUBLEIN   number_of_double_inputs
SHOW          yes or no

(...)

EOF
```

VERSION is the version number of Onika under which this file was saved. This parameter is used by Onika to automatically update old files.

RMODFILE is the name of the task parameter file of a given task in the configuration (defined in Appendix A.1). *RMODFILE* is the start of the "block" of lines needed to describe a task in a configuration. There is one "block" of these lines for each task in the configuration.

COORDS is the (x, y) location of the upper left corner of the task on the canvas, and is the second line in the "block."

INLEFT, the third line in the "block," determines the directionality of the task's icon. If its value is "yes," then inputs are to be displayed on the left side of the icon. Otherwise, inputs are to be displayed on the right side of the icon.

FREQ gives the frequency of the task, and is the fourth line in the "block." This value overrides any value given in the task's parameter file, and is ignored if the task is aperiodic (though the line must still exist). Unlike the task parameter file, this line *cannot* be replaced with *PERIOD* instead.

RTPU is the last line in the "block." It lists the name of the RTPU on which the task should run. If the task was never spawned on an RTPU before saving the configuration, the value of this line will be "unknown."

LOCUS signifies the start of a connection locus information "block." All locus blocks follow the last task block. There is one "block" for each state variable used in the configuration. The value of the locus line is the name of the state variable.

COORDS is the (x, y) location of the upper left corner of the locus on the canvas, and is the second line in the "block."

CONNS is the third line in the block, and tells how many tasks are connected to the locus.

NUMINCONST, **NUMOUTCONST**, **NUMINVAR**, **NUMOUTVAR**, **NUMDOUBLEIN**: Lines four through eight deal with number of *INCONSTs*, *OUTCONSTs*, *INVARs*, *OUTVARs*, and *INBOTHs* related to the locus. These lines (along with *CONNS*) may be removed from future versions of Onika.

SHOW is the last line in the block., and tells whether the locus should have its connections drawn or not.

EOF should be the last line in the file.

A.3 Upper-level Icon (.*onk*) file

The upper-level icon file (*base_directory/dict/xxx.onk*, where "onk" is short for *onikon*, the internal name of an upper-level icon in Onika) holds all of the information needed for an icon.

The format of an icon file is as follows:

```
VERSION      version number of Onika in which the file was saved

ONIKON       icontype
DESC         description
UI           list of ui element values for ui item i or none
UI           list of ui element values for ui item 2

(...)

PICT         first row of icon picture definition characters

(...)

PICT         thirty-second row of icon picture definition characters

EOF
```

VERSION is the version number of Onika under which this file was saved. This parameter is used by Onika to automatically update old files.

ONIKON describes the type of icon; e.g., "Complete job," "Cartesian object," "Joint Application," etc. This name must exactly match one of the types defined in *$HOME/.OnikaUpperSetUp.dta* (or *$ONIKA_DIR/prefs/OnikaUpperSetUp.dta*, if the former does not exist).

DESC gives a description of what the icon does. Currently, Onika does not use this field, although it has to be in the file.

UI is either "none" (for non-object icons) or is a list of values for the object. In the latter case, the number of UI lines is equal to the number of data items defined for this type of *ONIKON*. The number of elements on each line is equivalent to the number defined for this particular data item in *$HOME/.OnikaUpperSetUp.dta* (or *$ONIKA_DIR/prefs/OnikaUpperSetUp.dta*, if the former does not exist).

PICT lines define the picture centered in the icon. There are exactly 32 of these lines, and each line has 32 characters. The characters range from '0' to '9.' When converted to integers, these characters represent offsets into a color table defined internally to Onika. Currently, 0=*red*, 1=*green*, 2=*blue*, 3=*yellow*, 4=*orange*, 5=*purple*, 6=*grey*, 7=*white*, and 8=*black*.

EOF must be the last line in the file.

A.4 Application (*.appl*) file

The application file (*base_directory/appl/xxx.appl*) holds all of the information that Onika needs to recreate an application. Its format is as follows:

```
VERSION      version number of Onika in which the file was saved

APPLTYPE     conditional or parallel

FLOW
```

Page 149 of 198

ABBINC_0125905

```
ROUTINE       name of icon
MARKER        yes or no
SYNCHTAG      tagname
WHILETAG      tagname
WHILETEST     yes or no
ONIKON        icontype
DESC          description
UI            list of ui element values for ui item 1 or none
UI            list of ui element values for ui item 2

(...)

PICT          first row of icon picture definition characters

(...)

PICT          thirtysecond row of icon picture definition characters

(...)

ICONHOLE

(...)

ENDFLOW

FLOW

(...)

ENDFLOW

(...)

EOF
```

VERSION is the version number of Onika under which this file was saved. This parameter is used by Onika to automatically update old files.

APPLTYPE tells whether the application is conditional or parallel. Although the distinction is meaningless if only one flow is involved, the line must nevertheless be in the file.

FLOW marks the beginning of one of the application flows. The first *FLOW* follows *APPLTYPE*, whereas subsequent *FLOW* markers follow the *ENDFLOW* marker of the preceding flow. All icons and "holes" encountered between a *FLOW* and its *ENDFLOW* are considered part of that flow

ROUTINE gives the name of the icon and marks the beginning of lines of code which define an icon. For instance, if the value is *xxx*, then the *.onk* file for this icon is in *some_dir/dict/xxx.onk*, and (if an action icon) the routine to be run when this icon is encountered during execution is either *some_dir/conf/xxx.conf* or *some_dir/appl/xxx.appl*.

MARKER tells whether or not a breakpoint is set after the icon.

SYNCHTAG is optional and gives the name of the synchronicity tag of the icon.

WHILETAG is optional and gives the name of the while-loop tag of the icon..

WHILETEST is compulsory if *WHILETAG* is included, and must follow *WHILETAG* in such a case. It tells whtether or not the icon's job's return value should be analyzed to test loop conditions.

ONIKON describes the type of icon; e.g., "Complete job," "Cartesian object," "Joint Application," etc. This name must exactly match one of the types defined in *$HOME/.OnikaUpperSetUp.dta* (or *$ONIKA_DIR/prefs/OnikaUpperSetUp.dta*, if the former does not exist).

DESC gives a description of what the icon does. Currently, Onika does not use this field, although it has to be in the file.

UI is either "none" (for non-object icons) or is a list of values for the object. In the latter case, the number of UI lines is equal to the number of data items defined for this type of *ONIKON*. The number of elements on each line is equivalent to the number defined for this particular data item in *$HOME/.OnikaUpperSetUp.dta* (or *$ONIKA_DIR/prefs/OnikaUpperSetUp.dta*, if the former does not exist).

PICT lines define the picture centered in the icon. There are exactly 32 of these lines, and each line has 32 characters. The characters range from '0' to '9.' When converted to integers, these characters represent offsets into a color table defined internally to Onika. Currently, 0=*red*, 1=*green*, 2=*blue*, 3=*yellow*, 4=*orange*, 5=*purple*, 6=*grey*, 7=*white*, and 8=*black*. The thirty-second *PICT* line marks the end of the lines of code which define a single icon in the application.

ICONHOLE means that, at this point in the flow, there is a "hole" rather than an icon, and no icon data should be loaded for this position. The next icon encountered will be loaded into the next position in the application.

ENDFLOW signals the end of a flow within an application. It is either followed by another *FLOW* introduction or *EOF*. There is exactly one *ENDFLOW* marker for each *FLOW*.

EOF must be the final line in the file.

A.5 Preference file (*$HOME/.onika*)

This file is optional. If it exists, then Onika will use the preferences stored within it. (If only the upper level is used, then a preference file will automatically be created when the user exits Onika.) The format of the file is as follows:

```
VERSION       version number of Onika in which the file was saved

LINELEN       length of task names before wraparound
MINSIZE       minimum height of a task icon
MINPINSEP     minimum distnce between task icon pins
STEPSIZE      step size used if icon needs to increase to next size
PINLENGTH     length of pins
CONNSEP       separation between connections leaving a task
INCONST       yes or no
OUTCONST      yes or no
INVAR         yes or no
OUTVAR        yes or no
LOCISIZE      visual size of loci on screen
HITTOL        the hit tolerance within Onika's middle level
```

```
RATE          frequency or period
LEXI          x y h w
CONF          x y h w
STAT          x y
LOCUS         x y
DICT          x y h w
APPL          x y h w
MAIN          x y
BUTTON        x y
STATINFO      list of status information names

EOF
```

VERSION is the version number of Onika under which this file was saved. This parameter is used by Onika to automatically update old files.

LINELEN is the number of characters of a task's name that will be displayed in the icon before wraparound occurs.

MINSIZE is the minimum height of an icon in pixels.

MINPINSEP is the minimum distance between two pins on the side of an icon. If this distance cannot be maintained between pins, the Onika will increase the size of the icon by the value of the *STEPSIZE* field. This process continues until the all pin distances are at least the value of the *MINPINSEP* field.

STEPSIZE is used to determine how much larger an icon should be made to try to fulfill the *MINPINSEP* criteria.

PINLENGTH is the length of the pins. Note that, as the pins get longer, the "box" portion of the icon gets skinnier, since icons can only take up a certain amount of width (this value is hard-coded into Onika).

CONNSEP is used to keep connections from running into each other as they go around their source icon. All connections leaving the same icon side will be separated by the distance in the *CONNSEP* field.

INCONST, OUTCONST, INVAR, OUTVAR: These fields describe whether or not loci having these characteristics have their connection drawn. "Yes" takes preference over "no;" for instance, if *INCONST*s are not to be shown, but *INVAR*s are, then a loci having both an *INCONST* and an *INVAR* will have its connection shown.

LOCISIZE is simply the size of the radius of loci on the canvas.

HITTOL describes how close you have to come to a pin or locus to actually have it register as being touched. This can be as high as 10 if the user would otherwise have difficulty pinpointing something so small with a mouse.

RATE is used to determine whether the user prefers to view rates as frequencies or periods.

LEXI, CONF, STAT, LOCUS, DICT, APPL, MAIN, BUTTON: these fields give the absolute *x* and *y* positions of the task library, configuration canvas, RTOS status window, RTOS state variable values window, job dictionary, application workspace, Onika Command Window, and Onika Default Functions window, respectively. For the library, canvas, workspace, and dictionary, h and w give the dimensions of the scrollable portion of the window.

STATINFO contains a list of the names of the types of status information which the user wants presented to him or her. The information which can be displayed include *rmod* (module name), *rtpu* (rtpu name), *tid* (task ID), *S* (type of task: "P" for periodic, "Y" for synchronous, "A" for aperiodic), *crit* (criticality), *state* ("off," "cycle," "synch," "error," etc.), *cycle* (number of cycles sine the last status retrieval command was issued from Onika), *miss* (number of missed cycles), *ref-F* (desired frequency), *ref-T* (desired period), *mez-F* (average cycles/second executed), *mez-T* (measured period as detected by automatic profiling), *ptime* (profiling time), *tot-C* (total execution time used by task within profiling time), *tick* (clock tick of the RTPU on which the task is on), *min-U* (minimum RTPU utilization/cycle), *max-U* (maximum RTPU utilization/cycle), *avg-U* (average RTPU utilization/cycle), *min-C* (minimum RTPU time/cycle), *max-C* (maximum RTPU time/cycle), *avg-C* (average RTPU time/cycle), and *warn* (number of warnings issued since last status retrieval command was issued by Onika).

EOF must be the last line in the file.

A.6 Onika Default Functions File (*$HOME/.onika.buttons*)

This file is optional. If it exists, then, at start-up, Onika creates system call buttons based on the information in this file. The amount of buttons that can be created in this fashion is limited only by memory. The format is as follows:

```
VERSION       version number of Onika in which the file was saved

BUTTON        button name
FUCTION       command to be issued
SPAWNTTY      yes or no

BUTTON        button name
FUCTION       command to be issued
SPAWNTTY      yes or no

(...)

EOF
```

VERSION is the version number of Onika under which this file was saved. This parameter is used by Onika to automatically update old files.

BUTTON is the name that will appear on the button. The first button defined follows *VERSION*; other *BUTTON*s follow the *SPAWNTTY* line of the previous button "block."

FUNCTION is the command that will be issued when the button is pressed. It does not need an ampersand at the end, as Onika will supply that itself if one is missing.

SPAWNTTY indicates that this command should be run through a newly-generated *xterm* rather than on the default shell (the one in which Onika was launched). When the command finishes execution, the xterm is automatically killed.

EOF must be the last line in the file.

A.7 The Syntax Definition File (*.OnikaUpperSetUp.dta*)

Onika looks for this file in the directory *$HOME*. If it is not found, Onika uses

$ONIKA_DIR/prefs/OnikaUpperSetUp.dta instead.

This file sets up the syntax of the upper level of Onika. The user should not change the names or definitions of "Application start," "Application end," "Complete job," or "Complete application." For new actions requiring objects, make certain that the word "job" or "application" appears as part of the *ONIKON* field, preceded by the name of the object required. If "Xxx job" exists, then "Xxx application" should also be specified, and vice-versa, unless there's some overwhelming reason why one or the other should not be permitted. In general, the fields of "Xxx job" and "Xxx application" should always be the same, except for the information in the *ONIKON* field itself.

The format of the file is as follows:

```
VERSION        version number of Onika in which the file was saved

ONIKON         typename
LEFTEDGE       edge number
RIGHTEDGE      edge number

ONIKON         typename
LEFTEDGE       edge number
RIGHTEDGE      edge number
VARNAME        name of this data item
DATATYPE       int, double, or float, etc...
NELEMS         number of elements for this data item
VARNAME        name of this data item
DATATYPE       int, double, or float, etc...
NELEMS         number of elements for this data item

(...)

EDGENUM        0
RGB            r g b
NUMVERTS       number of vertices
VERTICES       list of relative x/y movements

EDGENUM        1
RGB            r g b
NUMVERTS       number of vertices
VERTICES       list of relative x/y movements

(...)

EOF
```

VERSION is the version number of Onika under which this file was initially used. This parameter is used by Onika to compensate for older files.

ONIKON is the name given to the type of icon being defined.

LEFTEGDE defines which edge information is used on the left side of the icon. The number refers to an *EDGENUM* block defined later in the file.

RIGHTEDGE is similar to **LEFTEDGE** but defines the right edge of this type of icon.

Page 154 of 198

ABBINC_0125910

VARNAME, DATATYPE, NELEMS: *VARNAME* is used only when the type is an object, in which case *DATATYPE* and *NELEMS* must follow it (in that order). These, in turn, may be followed by another *VARNAME-DATATYPE-NELEMS* block, since an object is not limited in the number of data items it can have. *VARNAME* indicates the name of this data item (used when the user is prompted during the session for values), *DATATYPE* indicates the type of information defined (*string*, *line*, *char*, *byte*, *float*, *double*, *long*, *int*, or *short*), and *NELEMS* indicates how many elements are in the array.

EDGENUM is the identification number assigned to an edge. This number must be unique with respect to other defined edges. The numbering need not be sequential nor contiguous.

RGB is the color of the edge given as a red-green-blue vector. *r*, *g*, and *b* range from 0 to 255.

NUMVERTS can loosely be described as the number of "zigs" or "zags" the edge of an icon makes.

VERTICES gives a *NUMVERTS*-long list of *x*/*y* pairs which tell Onika how to draw the icon's edge. Each pair is a relative offset from the previous position of the pen. The starting position is 16 pixels away from the closest edge of the main icon picture, at a height level equivalent to the top of the icon. A line will connect the last offset to a point 16 pixels away from the closest edge of the main icon picture, at a height level equivalent to the bottom of the icon. So, the line *VERTICES -5 5 5 5* would cause a line to be drawn from the starting position to a point five pixels to the left and five pixels down. The next line would be five pixels to the right and five pixels down from that new position, and finally that last position would be connected with the ending point automatically.

EOF must be the last line of the file.

A.8 Start and Stop Icon Files

The files *$ONIKA_DIR/prefs/OnikaStart.dta* and *OnikaStop.dta* are not like the other files listed in this appendix. They do not have a version number, nor are they based on the Chimera library file-formats. These files define the appearance of the "start" and "stop" icons of the upper level of Onika.

Both files consist of 32 lines having 32 characters each. These characters range from '0' to '9,' which, when interpreted as integers represent offsets into a color table. Currently, 0=*red*, 1=*green*, 2=*blue*, 3=*yellow*, 4=*orange*, 5=*purple*, 6=*grey*, 7=*white*, and 8=*black*. Thus, the definition of the icon pictures is similar to that found in the *PICT* fields of the *.onk* files (Appendix A.3), but the keyword *PICT* does not precede the lines, nor are there any other lines in the files.

Appendix B: Mouse Clicks

This appendix gives the user tables of what the various mouse and key combinations mean in the configuration canvas, task library, job dictionary, and application workspace. Note that the "mouse button" is pressed with the "key" already down.

B.1 The Task Library

Table 5: The Task Library

mouse button	key	location of mouse	action
left	*(doesn't matter)*	over icon	selects the icon, allows dragging
left	*(doesn't matter)*	over empty space	deselects any selected icon in the current library
middle	*(none)*	over icon	shows task parameters
middle	"Shift"	over icon	displays C file
middle	"Control"	over icon	displays task parameter file

B.2 The Configuration Canvas

Table 6: The Configuration Canvas

mouse button	key	location of mouse	action
left	*(none)*	over icon	selects the icon, allows dragging
left	*(none)*	over locus	allows dragging of locus
left	*(doesn't matter)*	over empty space	deselects any selected icon in the configuration
left	"Shift"	over icon	toggles task activity — if task is in error state, tries to clear task
left	"Shift"	over locus	toggles whether the locus' connections are shown
middle	*(none)*	over icon	shows task parameters
middle	(none)	over locus	shows connection information
middle	"Shift"	over icon	displays C file
middle	"Shift"	not over icon	checks configuration
middle	"Control"	over icon	displays task parameter file
right	*(none)*	anywhere	if an icon is selected in the library, this places it on the canvas

Table 6: The Configuration Canvas

mouse button	key	location of mouse	action
(none)	"Delete" or "Back Space"	anywhere	cuts the currently selected icon
(none)	"C"	anywhere	copies the currently selected icon
(none)	"V"	anywhere	pastes an icon from the buffer
(none)	"R"	anywhere	reverses the I/O direction of the currently selected icon

B.3 The Job Dictionary

Table 7: The Job Dictionary

mouse button	key	location of mouse	action
left	*(none)*	over icon	selects the icon, allows dragging
left	*(doesn't matter)*	over empty space	deselects any selected icon in the current library

Table 7: The Job Dictionary

mouse button	key	location of mouse	action
left	"Shift"	over job icon	swaps job configuration represented by the icon into current configuration if middle level programming is active
left	"Shift"	over application icon	opens application represented by the icon
middle	*(none)*	over action icon	shows icon name and types
middle	*(none)*	over object icon	shows object values, which can be edited
middle	"Shift"	over icon	opens icon picture editor

B.4 The Application Workspace

Table 8: The Application Workspace

mouse button	key	location of mouse	action
left	*(none)*	over icon	selects the icon

Page 160 of 198

Table 8: The Application Workspace

mouse button	key	location of mouse	action
left	*(doesn't matter)*	over empty space	deselects any selected icon in the current library
left	"Shift"	over job icon	swaps job configuration represented by the icon into current configuration if middle level pro-gramming is active
left	"Shift"	over appli-cation icon	opens appli-cation repre-sented by the icon
middle	*(none)*	over action icon	shows icon name and types
middle	*(none)*	over object icon	shows object values, which can be edited
middle	"Shift"	over icon	opens icon picture editor
right	*(none)*	anywhere	if an icon is selected in the dictionary, this places it on the canvas
(none)	"Delete" or "Back Space"	anywhere	cuts the cur-rently selected icon
(none)	"C"	anywhere	copies the currently selected icon

Table 8: The Application Workspace

mouse button	key	location of mouse	action
(none)	"V"	anywhere	pastes an icon from the buffer

Appendix C: How to Create Tasks

Onika requires certain modules to act in certain ways. Of particular note are modules which signal the end of a job. This section describes the signals which Onika expects from the modules, as well as user input specifications for modules. It is intended for users who write the code for modules, and who are experienced with signal handling in the real-time operating system. Readers of this section are assumed to be familiar with the Chimera 3.1 User's Manual.

C.1 Trigger Modules

In Onika, we define a *trigger module* as a module which signals the end of a job. For instance, in a joint motion job, the trajectory module is the trigger module, since it is the module which decides whether or not the destination has been reached, and the job is finished. There should only be one trigger module per configuration or job.

Typically, the trigger module is the module which defines the type of job a configuration would make. For instance, the presence of a joint trajectory module indicates that the configuration is a joint job requiring a joint object (i.e. the trajectory endpoint in joint coordinates), whereas the presence of a Cartesian trajectory module indicates that the configuration should be regarded as a Cartesian job, requiring a Cartesian object (i.e. the trajectory endpoint in Cartesian coordinates). Of course, some trigger modules do not require any object to run; gripper modules and trackball modules are a good example of these. Trigger modules which do require an object should have the *ONIKA* fields in their *.rmod* files reflect this (see Appendix A.1).

Trigger modules should not turn themselves off by returning *SBS_OFF.* Instead, they should use *SBS_FINISHJOB(x)*, which sends a signal that they are finished before implementing the *xxxOff()* routine. The actual signal sent using this command is *SBS_SIG_FINISH* | *x*, where x generally gives some indication as to which flow path in an application should be taken next, or whether looping should be performed. Here is an example from a periodic trigger module's C code:

```
#define GO_TO_FLOW(i) SBS_SIG(SBS_SIG_END+i+2)
#define WHILECONTINUE SBS_SIG(SBS_SIG_END+1)

/* ... */

typedef struct {
   /* normal local variables here */
   /* ... */
} trajectLocal_t;

/* ... */

trajectOn(stask,local)
trajectLocal_t    *local;
sbsTask_t         *task;
{
   /* Do normal "on" number-crunching */
}

trajectCycle(stask, local)
trajectLocal_t    *local;
sbsTask_t         *task;
{
      /* do normal cycle number-crunching */
      /* ... */
      if trajectory_is_finished()
      {
         return SBS_FINISHJOB(GO_TO_FLOW(0));
      }
      /* ... */
   return I_OK;
}

/* ... */
```

Once the trigger module reaches the end of the trajectory, it returns *SBS_FINISHJOB(GO_TO_FLOW(0))*. The module is deactivated by Chimera (as if SBS_OFF had been returned, but the signal *SIG_FINISH* | *GO_TO_FLOW(0)* is sent to Onika as well. Onika will receive the signal and, if the module is operating within the context of an application, will route the application onto flow 0. If the value passed to *GO_TO_FLOW()* had been *i*, where *i* was unequal to 0, Onika will route the application onto flow *i* if more than one choice of flows exists in the next high-level icon in the application. (If only flow 0 is available, that flow will be taken regardless of the value of *i*.) Flows are numbered from 0 to *MAXFLOW*-1. (Currently, *MAXFLOW* is defined as 3 in Onika.) By convention, flow 0 indicates that the modules finished in a typical state, and the default flow should therefore be taken. If *WHILECONTINUE* had been sent as well (i.e. *SBS_FINISHJOB(GO_TO_FLOW(0) | WHILECONTINUE)*, any loops for which this was the test configuration would have continued looping.

Non-trigger modules should not use *SBS_FINISHJOB(x)*. Passive modules (i.e. modules for whom their deactivation would not affect the rest of the system and does not indicate the end of a job's execution) should instead return *SBS_OFF* if they must turn themselves.

C.2 *SBS_OFF* vs. *errInvoke()*

Programmers may be tempted to have their *xxxCycle()* or *xxxSync()* routines return *SBS_OFF* rather than returning *SBS_FINISHJOB(x)*, or to return *SBS_OFF* to deal with errors noticed in the *xxxCycle()* routine. Such a design philosophy can lead to bad consequences when software is shared amongst various users.

If an error is encountered (for instance, "Joints exceeding limits"), then the proper way to deal with this is to call an *errInvoke()*. This will send the task into the *xxxError()* routine, allowing the condition to be dealt with as appropriate. If *xxxError()* cannot deal with the error, and returns *SBS_ERROR*, then an error signal will be sent to Onika automatically. If *SBS_CONTINUE* is returned by the *xxxError()* routine instead (indicating the problem has been fixed), then execution will continue as normal, and Onika will not be signalled. This allows the Onika user to remain informed about unrecoverable errors, but to be unburdened with those that can be automatically fixed.

If a stopping condition is encountered (for instance, "Endpoint reached"), then Onika should be signalled with the *SBS_FINISHJOB(x)* return option, where *x* contains the appropriate flow information signal. This allows Onika to move on to the next job at the upper level. The stopping task will be turned off automatically by Chimera regardless of whether it is in an application or not, removing the need for the task to return *SBS_OFF*. Trigger modules are the only modules which should return *SBS_FINISHJOB(x)*. When non-trigger modules need to turn off, they should be considered as being in error, and appropriate measures should be taken.

There is one case, however, when a task can turn itself off by returning *SBS_OFF* with no problems. This is when the task is *passive*; that is, it does not affect the operation of the other modules in its configuration, nor the outcome of the job. A good example would be a display task which sent current joint variable values in a robot motion job to an external robot CAD-display viewer. If, for some reason, a user running an application containing this job decides to save time and not run the robot viewer, then the display task can and should turn itself off (i.e. return *SBS_OFF*) when it fails to connect to the external robot viewer. It is true that this action will cause a signal to be sent to Onika; however, Onika will ignore *SBS_SIG_OFF* when an application is running. This removes the need for the programmer to remove the offending task from the job or rewrite the entire application, and will not cause a robot motion application to come to a halt just because a viewer used in one of its jobs isn't running. Note that passive tasks are *never* trigger tasks.

C.3 Signal Response

The following table illustrates how Onika responds to signals coming from a task on the real-time operating system.

Table 9: Signal Response in Onika

Signal sent	No application is executing	An application is executing
SBS_SIG_OFF	Onika updates status. In general, tasks should not turn themselves off, but invoke an error instead, unless the task is passive.	Ignored (though this will probably change). In general, tasks should not turn themselves off; they should invoke an error instead, unless the task is passive.
SBS_SIG_FINISH	Onika deactivates the task and updates the status information.	Onika deactivates the task, then moves to next job in application. If no flow information is sent with the signal, flow 0 will be assumed. If no loop information is included, any while loops will be exited.
SBS_SIG_ERROR	Onika updates status, reporting error.	Onika reports error. User has option of trying to clear error and continue, or abort.
SBS_SIG_WARNIN G	Ignored.	Ignored. Note that errors which are converted into warnings in the *xxxError*() routine therefore do not affect application execution.

Table 9: Signal Response in Onika

Signal sent	No application is executing	An application is executing
SBS_SIG_SHUTDOWN	The user is informed. If communication with the RTOS fails, Onika goes into non-RTOS mode.	Application aborts.
SBS_SIG_ILLEGAL	The user is informed.	Application aborts.
SBS_SIG_LEGAL	Ignored.	Ignored.
SBS_SIG(*x*)	Ignored	If next job has multiple flows, flow *x*-*SBS_SIG_END*+2 will be followed; otherwise, flow 0 will be followed. If the while-loop tag of the icon is the testing instance, then if *SBS_SIG_END*+1 is set, execution will commence at or after the other instance (depending on whether it is a top or bottom test); otherwise, the loop is exited and execution continues from the icon following the current icon.

C.4 User Input (the UI packet)

Occasionally, when a module is controlled in certain ways (for example, when it is activated), it will require user input. For instance, a trajectory module might require the endpoint of the trajectory, or a gripper module might require the amount of force to be applied when gripping. To get this information, the module creator should allow for two possible routes. First, the command line should be checked for the values. If the value list is incomplete, or if the values supplied are not within appropriate bounds, then a UI structure should be created, initialized with the proper values, and a UI request should be sent. An example follows:

```
int my_moduleOn(local,stask)
```

```
my_moduleLocal_t *local;
sbsTask_t *stask;
{
   char *head,*tail;
   int good_args=FALSE, *ptr_int, the_int, default_int = 2,
      some_int_max = 5, some_int_min = 0;
   float *ptr_float, the_float, default_float = 1.5708,
      some_float_max = 3.14159, some_float_min = 0.0;
   UI *ui;

   /* This module requires an int and a float to cycle*/
   if (strlen(stask->argptr) > 0)   /* There are arguments */
   {
      good_args = TRUE;
      head = stask->argptr;
      the_int = cmdiArgInt(&head, &tail, NULL, MAXINT);
      head = tail;
      if (the_int < some_int_min || the_int > some_int_max)
         good_args = FALSE;
      else if (*head)
      {
         the_float = cdmiArgFloat(&head,&tail,
               NULL,MAXFLOAT);
         head=tail;
         if (the_float < some_float_min ||
            the_float > some_float_max)
               good_args = FALSE;
      }
      else good_args = FALSE;
   }

   if (*head) /* Too many arguments */
      good_args = FALSE;

   if (!good_args) /* If args were bad in any way*/
   {
      ui = uiCreate(2,512);

      ptr_int = (int *) uiInt(ui,"The integer",
         default_int, some_int_min, some_int_max);
      ptr_float = (float *) uiFloat(ui,"The float",
         default_float, some_float_min, some_float_max);
      sbsUserInput(stask,ui);

      the_int = *ptr_int;
      the_float = *ptr_float;
      uiFree(ui);
   }
   /* ... now do the number-crunching here ...*/
   return I_OK;
}
```

When turned on, the module *my_module* looks for the values it requires in the arguments passed to it. If these arguments are in any way improper, then the module sends a UI request to Onika. The user supplies the values, and the execution can commence.

If user input is required for any task at the middle level, a UI request should be sent to Onika. Onika will parse the UI packet, and present it to the user in a readable format in a pop-up window. The user supplies values as appropriate and presses the "Send to RTOS" button. Onika checks the values the user has entered against the minimums and maximums specified (if any) for each data item requested. If the values are out of range, Onika will request that the user try again, and will redisplay the pop-up window in which the values can be entered. Onika does not send user input in the argument line of task control packets when used at the middle level, because the data values are not yet known.

If user input is required for a task in an executing application, and if the task is a trigger module, then the data will be passed from the appropriate object icon to the task in the argument line of the task control packet used to activate the task. In fact, the presence of the object icon is what indicates to Onika that there is information that can go into the command line arguments of the trigger task. If the argument line contains faulty data, or if the task is not a trigger module, then the task should send a UI request, which will be processed as is done in the middle level. Onika only sends user input in the argument line of trigger module task control packets, and then only when the task is being activated. Otherwise, user input is handled as in the middle-level case.

Note that Onika expects that task modules which require user input will try to parse their argument lines and, if this fails, will send out a UI request. Tasks which do not follow this format will not work properly under Onika. The types of user input which Onika understands and can process are *string*, *line*, *char*, *byte*, *float*, *double*, *long*, *int*, and *short*. Both scalars and vectors are supported.

Appendix D: Future Hypermedia Engine Code

```
/*          Copyright (c)1994 by Carnegie Mellon University */
static char _onika_id[] = "@(#)File <filename>, Onika 2.0, CMU 25-Feb-
94";

/************************** HISTORY ***************************/
/*          25-Feb-94:  Matthew W. Gertz (mwgertz) at CMU*/
/*                                  Created for Onika 2.0
*/
/*************************************************************/

typedef struct hlink {
            struct hlink *prev,*next;
            char resource[MAXFNAMELEN];
        char cached[MAXFNAMELEN];
} HLINK;

#define HLINK_NULL (HLINK *)0

HLINK *Hlink_header=HLINK_NULL,*Hlink_tailer=HLINK_NULL;
char OnikaTempDirectory[255];

char *getcacheHlink(resource)
char *resource;
{
    HLINK *hp=Hlink_header;

    while (hp)
    {
            if (strcmp(hp->resource,resource))
                hp=hp->next;
            else break;
    }
    if (hp) return hp->cache;
    else return (char *)0;
}

char *getresourceHlink(cache)
char *cache;
{
    HLINK *hp=Hlink_header;

    while (hp)
    {
            if (strcmp(hp->cache,cache))
```

```
                hp=hp->next;
            else break;
    }
    if (hp) return hp->cache;
    else return (char *)0;
}

/* Given a resource, this procedure creates a cache version and */
/* opens it. */
/* Resource are in the form path@address */

FILE *openHlink(resource,arg)
char *resource;
char *arg;
char *rcache;
{
    char *cache,*address,fname[MAXFNAMELEN],
                        nfname[MAXFNAMELEN],command[2*MAXFNAMELEN];
    FILE *fp;
    char *user,*host;

    cache=getcacheHlink(resource);
    if (cache)
    {
            fp=fopen(cache,arg);
            return;
    }

    strcpy(fname,resource);
    address = strchr(fname,'@');
    if (!address)
    {
            if ((fp=fopen(resource,arg))!=(FILE *)0)
                addentryHlink(resource,resource);
            return fp;
    }

    address[0]='\0';
    address++;

    user=getenv("USER");
    host=getenv("HOST");

    sprintf(nfname,"%s/onikaftp",OnikaTempDirectory);
    fp=fopen(nfname,"w");
    fprintf(fp,"user anonymous\n");
    fprintf(fp,"password %s@%s\n",user,host);
    fprintf(fp,"bin\n");
    fprintf(fp,"lcd %s\n",OnikaTempDirectory);
    fprintf(fp,"get %s\n",resource);
    fprintf(fp,"quit\n");
    fclose(fp);

    sprintf(command,"ftp -n -i %s < %s > /dev/null",address,nfname);
    system(command);
    sprintf(command,"rm -f %s",nfname);
    system(command);
    sprintf(command,"/tmp/onika/%s",fname);
    if ((fp=fopen(command,arg))!=(FILE *)0)
            addentryHlink(resource,command);
    else
            DoAlert("ONIKA_STR_BADHLINK",resource);
    return fp;
}

void addentryHlink(resource,cache)
char *resource,*cache;
{
   HLINK *node;
   MALLOC_ERROR(node=(HLINK *)sizeof(HLINK));
   strcpy(node->resource,resource);
   strcpy(node->cache,cache);
   if (!Hlink_header)
   {
        Hlink_header = node;
        Hlink_tailer = node;
        node->next=(HLINK_NULL);
        node->prev=(HLINK_NULL);
   }
   else
   {
        node->next=Hlink_tailer->next;
        node->prev=Hlink_tailer;
        Hlink_tailer->next=node;
        Hlink_tailer=node;
   }
}

int initializeHlink()
{
    char command[2*MAXFNAMELEN];

    sprintf(OnikaTempDirectory,"/tmp/onika%d",getpid());
    sprintf(command,"mkdir OnikaTempDirectory");
    system(command);
}

int flushcacheHlink(resource)  /* flushes changes to buffer */
char *resource;
{
    char *cache,*address,nfname[MAXFNAMELEN],
              fname[MAXFNAMELEN],command[2*MAXFNAMELEN];
    FILE *fp;
    char *user,*host;

    cache=getcacheHlink(resource);
    if (!cache) return 0;
    strcpy(fname,resource);
    address = strchr(fname,'@');
    if (!address)
    {
            return 1; /* 'twas on the local filesystem */
    }

    address[0]='\0';
    address++;

    user=getenv("USER");
    host=getenv("HOST");

    sprintf(nfname,"%s/onikaftp",OnikaTempDirectory);
    fp=fopen(nfname,"w");
    fprintf(fp,"user anonymous\n");
    fprintf(fp,"password %s@%s\n",user,host);
    fprintf(fp,"bin\n");
    fprintf(fp,"lcd %s\n",OnikaTempDirectory);
    fprintf(fp,"put %s\n",cache);
    fprintf(fp,"quit\n");
    fclose(fp);

    sprintf(command,"ftp -n -i %s < %s > /dev/null",address,nfname);
    system(command);
    sprintf(command,"rm -f %s",nfname);
    system(command);

}

FILE *getlistHlink(resourcesite)  /* returns a filelist from a
resource site */
                                                  /* by putting it in
a file */
char *resourcesite;
{
    char *address,command[2*MAXFNAMELEN],
            nfname[MAXFNAMELEN],fname[MAXNAMELEN];
    FILE *fp;
    char *user,*host;

    address = strchr(resource,'@');
    if (!address)                    /* local file */
    {
            sprintf(command,"/usr/bin/ls %s/%s > %s/lstemp",
                        resourcesite,OnikaTempDirectory);
            system(command);
    }
    else
    {
            address[0]='\0';
            address++;

            user=getenv("USER");
```

Page 174 of 198

```
        host=getenv("HOST");

        sprintf(nfname,"%s/onikaftp",OnikaTempDirectory);
        fp=fopen(nfname,"w");
        fprintf(fp,"user anonymous\n");
        fprintf(fp,"password %s@%s\n",user,host);
        fprintf(fp,"cd resourcesite\n");
        fprintf(fp,"ls\n",cache);
        fprintf(fp,"quit\n");
        fclose(fp);

        sprintf(fname,"%s/lstemp",OnikaTempDirectory);
        sprintf(command,"ftp -n -i %s < %s > %s",
                    address,nfname,fname);
        system(command);
        sprintf(command,"rm -f %s",nfname);
        system(command);
    }

    fp=fopen(fname,"r");
    return fp;
}
```

Appendix E: User Testing

E.1 Checklist

This is the checklist used for the user testing (modified slightly when, for instance, the non-pictures version of Onika was used):

Name: ______________________

Gender: ______________________

Subject Number: ______________

Date: ______________________

Time: ______________________

Naming scheme: ______________

☐ Make certain this name has not been used before!

Last grade completed: HS FR SO JU SE MS PhD

Job description: ____________________________

What kind of computers/OS has subject used in the past?

____

Has subject ever written a program before? ....Yes/No

Does subject consider self a programming expert? ....Yes/No

Has subject used a robot before? ....Yes/No

Does subject consider self a robotics expert? ....Yes/No

Before the subject enters the laboratory:

☐ Clean up the workspace and the screen

Page 177 of 198

- ☐ Make sure subject will have view of PUMA C
- ☐ Set up mock-up
- ☐ Turn on the air supply to the gripper

From yow.ius.cs.cmu.edu:

- ☐ Run “otest” in all of the windows.
- ☐ Log in to e5.ius.cs.cmu.edu
- ☐ Run “otest” in the e5 window
- ☐ On e5: % cd base/bin
- ☐ On e5: % chim
- ☐ On e5: ex calibrate C (if necessary)
- ☐ On e5: ex calibrate B (if necessary)
- ☐ On e5: <Go_cst
- ☐ On e5: go all
- ☐ Move PUMA B to (0, -pi/2, pi/2, 0, 0, 0) if not already there
- ☐ Move PUMA C to (0.0114, -0.226, 2.35, -0.0371, 1.06, 1.61):

  <power_up; <tri_above; off
- ☐ Open the gripper on PUMA C:

  on gripC; off
- ☐ Calibrate on the point (-0.0417, -0.0328, -0.999, 0.0829, 0.999, -0.0332, 0.781, 0.159, -0.21):

  <tri_to; off; off fwdkin; off invkin
- ☐ Move PUMA C to (0, -pi/2, pi/2, 0, 0, 0):

  <tri_above; off; <home; off
- ☐ Close gripper:

on gripC; off

- ☐ Restart Chimera: quit, quit, chim, <Go_cst, go all
- ☐ Set up Chimera network connection (network)
- ☐ On yow: % cd base
- ☐ On yow: % Onika -up -chim -log ***or*** % ndOnika -up -chim -log
- ☐ Enter log filename according to convention given above (*.raw)
- ☐ Close and hide all windows except for controls and dictionary
- ☐ Move "Move to" picture to be next to "move above"
- ☐ Make sure all dictionary icons can be seen
- ☐ Open "go_home.appl"
- ☐ Make sure all three windows are positioned properly

☐ Put out "how to" sheet

☐ Move joystick and buttons to within reach

After subject enters the laboratory and sits at yow:

☐ Briefly describe experiment and ask subject to sign consent sheet

☐ Record demographics

☐ **"I am going read some background information and instructions for this test to you. From this point on, I am not allowed to answer any questions until the test has concluded.**

☐ **"We are studying how a person would go about designing a program to assemble a small structure using a robot. These are the four boxes that make up the structure. These boxes which you will have the robot assemble are sitting on that table there *(point to it)*, but, for the purposes of demonstration, I will use this mock-up of the**

boxes when explaining this study to you. The box with the wavy lines will remain in place the entire time, and you will program the robot to place the other boxes on top of it in the correct order. When stacked in the correct order, they will make a structure that looks like this *(indicates stacked demo boxes and paper)*, and the light on the topmost box will light up when the box is touched.

☐ **"This program, called Onika, allows you to create programs for a robot by combining pictures of that robot with pictures of the objects manipulated by the robot in order to make a story which can be run on the robot. You can think of these pictures as pieces of a jigsaw puzzle, where each piece can be used in the puzzle more than once, and all of the pieces go one after the other. We want you to use Onika to create a program from these pictures that the robot can read in order to assemble the structure. The robot you will use is the one with the ribbon tied to it. It is a PUMA 560, which has six joints, as well as a hand -- also called a 'gripper' -- which can open and close.**

☐ **"The pictures available for your use in creating this story program are located in this dictionary *(point to the dictionary)*. I'll describe what each one does, using this small mock-up of the boxes to demonstrate: *(The test administrator will point to the appropriate icon as each one is mentioned, and pantomime, using the mock-up, as necessary.)***

☐ **"First, you should note that the edges of all of the pictures have certain colors and shapes. These edges are designed to help you determine which pictures can be placed next to each other. The actual colors and shapes of the edges are otherwise irrelevent. For instance, we might have made all of these red edges purple instead, or made all of these yellow edges blue.**

☐ **"The colors and shapes don't matter, except that the color of the right edge of a picture must match the left edge of the picture which you want to follow it, and their shapes must fit together like pieces of a jigsaw puzzle. For instance, since this picture ends up with a yellow triangular edge, it can be followed by any picture which start out with a yellow triangular edge. Red attaches to red, yellow attaches to yellow, and green attaches to green. You should not**

attach any other meaning to the colors.

☐ **"This picture causes the robot to move to a box's original position or destination position. You'll use it when you are moving down to pick up a box from its original position, or when placing a box in its destination position. The original position of a box is the place where it sits at the beginning of the task, and the destination position is the place where it is supposed to sit in the structure.**

☐ **"This picture causes the robot to move to a location above or near a box. You'll use this picture whenever you need to move to a spot above any box or above the structure. It's very important to move to a spot above a box or structure before moving to or from a box's original or destination position. If you don't, you might hit the box or the structure sideways, potentially damaging the boxes. I will demonstrate.**

☐ **"Pretend my hand is the hand of the robot. If the robot moves directly to this box, it will hit it sideways. To eliminate this possibility, the robot should first move above the box using the 'move above/near' picture, before moving down to the box using the 'move to' picture. When the box is grasped, the robot should then move back above where the box used to be, again using the 'move above/near' picture, in case there are other boxes nearby which might be hit accidentally.**

☐ **"Similarly, if you then moved the robot directly to the destination point of the box, you would hit the structure sideways. To avoid this, the robot should first move above the structure, using the 'move above' picture, and then 'move to' the destination point of the box. After releasing the box, you then cause the robot to 'move above' the structure again before going to get the next box; otherwise, you might knock the newly-placed box off the structure.**

☐ **"Therefore, the proper motions involved in placing a box on the structure are as follows: move above, move to, move above, move above, move to, move above.**

☐ **"Of course, the robot must open and close its hand at the proper times, too. Whenever you run the program, the robot's hand will start out closed, as it is now. This 'open gripper' picture opens the**

robot's hand. Remember that the robot's hand must be open before moving down to pick up any of the boxes, allowing the hand to slide around the box; otherwise, the hand will hit the box, and may damage it.

☐ **"Once the hand is around the box, you should use this 'close gripper' picture to cause the robot to close its hand on the box. You can then place the box on the structure. When the box has reached its destination on the structure, you should then use the 'open gripper' picture again to release the box.**

☐ **"When we combine the gripper pictures with the motion pictures, the resulting commands to place a box on the structure are as follows: open gripper, move above, move to, close gripper, move above, move above, move to, open gripper, move above. Programming the robot in this way greatly reduces the chances of the robot hitting any boxes accidentally.**

☐ **"All of the pictures mentioned so far are actions; that is, they cause the robot to do something. All actions in this dictionary start out red, but not all of them end up red. (Again, let me remind you that the actual color doesn't matter; the edges could just as well have been blue, so long as it was consistent.) Those that start out and end up red are completely defined actions, such as 'open the gripper.' The action pictures which don't end up red, however, need another picture to make them complete. For instance, to move the robot above a certain box, you need to specify which box to move above, since there are several boxes available. To specify which box to move above or to move to, we use these pictures which represent the locations of the boxes. These pictures store locations where the robot will move to or move above.**

☐ **"This line of pictures specifies the original and destination positions of the boxes which will be moved. Again, I will remind you that the original position of a box is the place where it sits at the beginning of the task, and the destination position is the place where it is supposed to sit in the structure. These pictures refer to the box with a triangle, these to the box with a square, and these to the box with a cross. Notice that their left edge is green, and fits like a puzzle piece with the 'Move to' picture. This shows you that these pictures** of the boxes are always used with the 'Move to' picture in order to make the robot move to the original or destination position for the specified box. It's important to note that, even when a box has been moved to some new location, the picture which refers to its original position still refers to where that box used to be, and the picture which refers to its destination position still refers to its designated location in the structure.

☐ "This next line of pictures specifies points above the original positions of each box. They interlock with the 'Move above/near' picture, which means that if you combine the 'Move above/near' picture with one of these pictures, you will move the robot to a point above the original position of the specified box. You should note two things. First, if a box is moved, its 'above' picture still refers to a spot above the box's original position. Second, since the box with the wavy lines will stay in place the entire time, and since the structure will be built on top of it, you can use its 'above waves' picture to mean 'above the structure.' This will allow you to move the robot to a spot above the structure before placing any box on it, keeping you from crashing sideways into it. 'Above waves' is exactly the same as 'above the structure.'

☐ "This last picture represents a 'no operation' action. It doesn't make the robot do anything. When the robot gets to this picture in your story, it will just skip over it and go to the next picture. You can use it to make certain sections of your program more readable by spacing them farther apart. For instance, you might insert one of these between the section of code which gets the box with a triangle, and the section of code which places it on the structure, just so it will be clearer to you later on where one begins and the other ends. However, you do not have to use this picture if you do not want to, as it will not affect the operation of the robot at all.

*(*WARM-UP*:)*

☐ "I will now demonstrate how to make a program. The program I will make will cause the robot to pick up the box with the triangle. When I am finished, I will ask you to try to make the same program so that you can practice making a program. First, I press the 'NEW' button. This gives me space on the computer to make a new

program. Just press 'OK' when you are asked what kind of program you want to make; the default is the proper answer. Here is the new program. It has a picture to tell you where it starts, and where it ends. It is otherwise empty.

☐ **"To place a picture into the program, you select it with the left mouse button in the dictionary, move the mouse cursor over the picture you want it to follow, and then press the right mouse button. To illustrate this, I will place the 'Move above/near' picture into the program. I select it with the left mouse button. I now move my cursor over the picture I want it to follow, and press the right mouse button. Because these two edges matched, I am allowed to place the picture, and it is inserted. Because these two edges didn't match, a hole was inserted. I will now fill up this hole. I select the 'above triangle's origin' picture with my left mouse. I now move my cursor over the 'move above/near' picture which I want it to follow. I press my right mouse button, and the hole is filled. This program now says, 'Move above the origin of the box with the triangle.' Note that the object follows the action, and that the action-object pair starts out and ends up red. This paper will remind you how to place a picture into the program if you need to be reminded.**

☐ **"After running this much of the program, the robot would be positioned above the triangle box. To pick it up, the robot now must move down to the box. I will therefore insert the 'Move to' picture *(does so)*, followed by the original position of the 'triangle' box.**

☐ **"If I want the robot to move to the box, though, I need to tell the robot to open its hand first, to avoid damaging the box. If I put the 'open gripper' picture here *(puts it at the end)*, then the hand would open after the robot had already crashed into the box. This is a mistake. To fix my mistake, I make sure that the faulty picture is selected by checking to see if it's highlighted -- a selected picture is highlighted with these dark edges and this bar across the top. If it wasn't highlighted, then clicking it with the left mouse button would select it as normal.**

☐ **"Once selected, I can then press the 'delete' key to remove it from the program. I can then insert the 'open gripper' picture in the proper place by making certain that it is selected in the dictio-**

nary, and then placing in the proper position by moving the cursor to the picture after which it should follow, and pressing the right mouse button. Notice that even though I made a mistake by leaving out the 'open gripper' picture at the beginning of the program, I didn't need to restart my work; I simply inserted the proper picture where it belonged. You can insert any needed picture anywhere in the program, provided that its left edge matches up properly with the picture it is to follow. You can also delete any picture from the program, and replace it with another picture if necessary, subject to the same rules about the edges matching up properly.

☐ **"Now I will insert the picture that closes the gripper. Notice that gripper picture starts and ends with a red edge, so that it needs no object picture to follow it. Finally, I will insert the required pictures to move back to the 'above triangle box' position.**

☐ **"Note how I can use the scroll bar to work with portions of the program which might not fit in the story program's window. To use the scroll bar, I move my cursor over this square, then hold down the left mouse button and move the mouse until the view is what I want. Please try this now *(lets user try the scroll bar).***

☐ **"Now I will run the program to see what it does. *(Runs the program.)* The program will execute in order from left to right. This program says, 'Open the hand, move above the original position of the box with the triangle, then move to that box, then close the hand, then move above the original position again.' In executing this story, the robot has picked up the box with the triangle painted on it, without running into the box from the side or making any other mistakes.**

☐ **"I will now remove the pictures. Again, notice that I can remove the pictures by selecting them with my left mouse button and pressing the 'Delete' key. Now, I will make the robot let go of the box, and cause it to move back to its 'starting' position with the hand closed. *(Does so.)***

☐ Move triangle box back to original position.

☐ **"Now, I would like you to rewrite the same program I just wrote; that is, just insert the pictures which cause the robot to pick**

up the box with the triangle. You don't have to create a new program this time, since there's already an empty one here for you to use. Please tell me when you are finished inserting the pictures you think are necessary, and then we will run the program to see if it works.

☐ Record time: ________

If subject fails:

☐ ☐ ☐ Explain why the failure occured.

☐ ☐ ☐ **"I will reset the robot, and you can fix your program and try again."**

☐ ☐ ☐ Run 'go_home.appl.'

☐ ☐ ☐ Move boxes back to original positions, and tell subject to proceed.

If subject succeeds:

☐ Record time: ________

☐ **"Now, I will remove this program, reset the robot, and let you create the entire program to assemble the structure. *(Does so.)***

☐ Move box back to original position.

☐ **"Your program should cause the robot to first assemble the structure, and then move to a point above the finished structure before ending. You have one hour from the time you press the "NEW" button to do so successfully. Any time you wish to try to run your story program, please let me know. You may modify and run your story as many times as it takes to get it right within the one hour limitation. I remind you that I am not permitted to answer any questions during the session, unless an error message appears. If an error message appears, please inform me immediately. This sheet summarizes what the mouse buttons do, and how to place pictures from the dictionary into the story program.**

☐ **"You may use these blocks in the mock-up to help you determine which actions you want the robot to take in order to assemble the structure. You may begin."**

☐ Record time: ________

If subject fails:

☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ Explain why the failure occured.

☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ **"I will reset the robot, and you can fix your program and try again."**

☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ Run 'go_home.appl.'

☐ ☐ ☐ ☐ ☐ ☐ ☐ Reset boxes, and tell subject to proceed.

When test concludes (either due to success or end of time limit):

☐ Record time: ________

☐ Save the current application (*.appl)

☐ **"Thank you very much for participating in our study. Here is your promised compensation. If you wish, we will be pleased to send you a summary of the results, including a discussion and analysis of the study."** ***(Get address if appropriate.)***

☐ Shut down everything.

☐ In yow, % proc <*raw data file*> to process, move, and back up data.

E.2 Raw Data

Here is a section of the raw data collected from one subject. It is abridged, as the entire data set for the subject would fill many pages:

```
Time        Event        Primary    PID       Secondary   SID
            Window
770908202   Start_Onika  none       0         none        0
            root
...
770909683   try_new_applnone        0         none        0
            appl
```

```
770909685    new_appl    Untitled-2  3577432     none        0
             appl
770909690    select_icon opengripC   3496920     none        0
             dict
770909692    insert_icon opengripC   3652992     Start       3684304
             appl
770909696    select_icon jmoveC      3494072     none        0
             dict
770909698    insert_icon jmoveC      3762528     opengripC   3652992
             appl
770909703    select_icon box2_aj     3509360     none        0
             dict
770909704    insert_icon box2_aj     3782232     jmoveC      3762528
             appl
770909708    select_icon cmoveC      3492592     none        0
             dict
770909712    insert_icon cmoveC      3785096     box2_aj     3782232
             appl
770909714    select_icon box2_bx     3498344     none        0
             dict
770909716    insert_icon box2_bx     3768784     cmoveC      3785096
             appl
770909731    select_icon closegripC  3495496     none        0
             dict
770909732    insert_icon closegripC  3773632     box2_bx     3768784
             appl
770909737    select_icon jmoveC      3494072     none        0
             dict
770909737    stopdrag_iconjmoveC     3494072     none        0
             dict
770909739    insert_icon jmoveC      3778480     closegripC  3773632
             appl
770909743    select_icon box2_aj     3509360     none        0
             dict
770909745    insert_icon box2_aj     3741504     jmoveC      3778480
             appl
770909749    select_icon jmoveC      3494072     none        0
             dict
770909759    insert_icon jmoveC      3748624     box2_aj     3741504
             appl
770909763    select_icon box1_aj     3506776     none        0
             dict
770909764    insert_icon box1_aj     3759856     jmoveC      3748624
             appl
770909774    select_icon cmoveC      3492592     none        0
             dict
770909776    insert_icon cmoveC      3813608     box1_aj     3759856
             appl
770909782    select_icon box2_ex     3499720     none        0
             dict
770909784    insert_icon box2_ex     3823440     cmoveC      3813608
             appl
770909789    select_icon opengripC   3496920     none        0
             dict
```

```
770909794    insert_icon opengripC   3832448     box2_ex     3823440
             appl
...
770910110    try_run_applnone        0           none        0
             none
770910110    run_appl    Untitled-2  3577432     none        0
             none
770910356    succeed_applUntitled-2  3577432     none        0
             none
```

E.3 Processed Data

Here is the processed data for the same subject:

```
Time required to request new application: 2 seconds
Elapsed time from NEW to success: 673 seconds
Total selection time: 242 seconds
Number of meaningful selections: 43
Average time per selection: 5.62791 seconds
Total number of selections: 43
Total number of insertions without selection: 0
Time required for insertions without selection: 0
Total number of all insertions: 43
Total insertion time (all): 124 seconds
Average insertion time: 2.88372 seconds
Number of deletions: 0
Total check time: 59 seconds
There were 1 attempts to run the application.
0 applications could not be run.
0 applications were aborted.
1 applications were successfully completed.
Total time used by successful applications: 246 seconds
Average time used by successful applications: 246 seconds
```

E.4 Timelined Data

The same subject's data is timelined to determine the duration of specific instances of events. Again, only an abridged example is offered.

```
Onika was started up at time t=0.
...
2 seconds later (at t=1483), the event new_appl occurred to item
      Untitled-2 (id = 3577432).
5 seconds later (at t=1488), the event select_icon occurred to
      item opengripC (id = 3496920).
2 seconds later (at t=1490), the event insert_icon occurred to
      item opengripC (id = 3652992) which followed/follows item
      Start (id = 3684304).
4 seconds later (at t=1494), the event select_icon occurred to
      item jmoveC (id = 3494072).
2 seconds later (at t=1496), the event insert_icon occurred to
```

```
     item jmoveC (id = 3762528) which followed/follows item
     opengripC (id = 3652992).
5 seconds later (at t=1501), the event select_icon occurred to
     item box2_aj (id = 3509360).
1 seconds later (at t=1502), the event insert_icon occurred to
     item box2_aj (id = 3782232) which followed/follows item
     jmoveC (id = 3762528).
4 seconds later (at t=1506), the event select_icon occurred to
     item cmoveC (id = 3492592).
4 seconds later (at t=1510), the event insert_icon occurred to
     item cmoveC (id = 3785096) which followed/follows item
     box2_aj (id = 3782232).
2 seconds later (at t=1512), the event select_icon occurred to
     item box2_bx (id = 3498344).
2 seconds later (at t=1514), the event insert_icon occurred to
     item box2_bx (id = 3768784) which followed/follows item
     cmoveC (id = 3785096).
15 seconds later (at t=1529), the event select_icon occurred to
     item closegripC (id = 3495496).
1 seconds later (at t=1530), the event insert_icon occurred to
     item closegripC (id = 3773632) which followed/follows item
     box2_bx (id = 3768784).
5 seconds later (at t=1535), the event select_icon occurred to
     item jmoveC (id = 3494072).
0 seconds later (at t=1535), the event stopdrag_icon occurred to
     item jmoveC (id = 3494072).
2 seconds later (at t=1537), the event insert_icon occurred to
     item jmoveC (id = 3778480) which followed/follows item
     closegripC (id = 3773632).
4 seconds later (at t=1541), the event select_icon occurred to
     item box2_aj (id = 3509360).
2 seconds later (at t=1543), the event insert_icon occurred to
     item box2_aj (id = 3741504) which followed/follows item
     jmoveC (id = 3778480).
4 seconds later (at t=1547), the event select_icon occurred to
     item jmoveC (id = 3494072).
10 seconds later (at t=1557), the event insert_icon occurred to
     item jmoveC (id = 3748624) which followed/follows item
     box2_aj (id = 3741504).
4 seconds later (at t=1561), the event select_icon occurred to
     item box1_aj (id = 3506776).
1 seconds later (at t=1562), the event insert_icon occurred to
     item box1_aj (id = 3759856) which followed/follows item
     jmoveC (id = 3748624).
10 seconds later (at t=1572), the event select_icon occurred to
     item cmoveC (id = 3492592).
2 seconds later (at t=1574), the event insert_icon occurred to
     item cmoveC (id = 3813608) which followed/follows item
     box1_aj (id = 3759856).
6 seconds later (at t=1580), the event select_icon occurred to
     item box2_ex (id = 3499720).
2 seconds later (at t=1582), the event insert_icon occurred to
     item box2_ex (id = 3823440) which followed/follows item
     cmoveC (id = 3813608).
```

```
5 seconds later (at t=1587), the event select_icon occurred to
      item opengripC (id = 3496920).
5 seconds later (at t=1592), the event insert_icon occurred to
      item opengripC (id = 3832448) which followed/follows item
      box2_ex (id = 3823440).
...
59 seconds later (at t=1908), the event try_run_appl occurred.
0 seconds later (at t=1908), the event run_appl occurred to item
      Untitled-2 (id = 3577432).
246 seconds later (at t=2154), the event succeed_appl occurred
      to item Untitled-2 (id = 3577432).
```

Bibliography

[1] J. S. Albus, H. G. McCain, and R. Lumia, "NASA/NBS standard reference model for telerobot control system architecture (NASREM)," NIST Technical Note 1235, 1989 Edition, National Institute of Standards and Technology, Gaithersburg, MD 20899, April 1989.

[2] M. Beretta, P. Mussio, and M. Protti, "Icons: Interpretation and Use," *IEEE Proceedings Workshop on Visual Languages 1986*, pp. 149-158.

[3] R. J. Brockmann, "The Unbearable Distraction of Color," *IEEE Transactions on Professional Communication*, vol. 34, no. 3, September 1991, pp. 153-159.

[4] T. B. Brown, "Completeness of a Visual Computation Model," Technical Report WUCS-93-53, Department of Computer Science, Washington University.

[5] S.- K. Chang, "Visual Languages: A Tutorial and Survey," *IEEE Software*, January 1987, pp. 29-39.

[6] S. -K. Chang (ed.), *Visual Programming Systems*, ©1990 Prentice Hall, Inc.

[7] L. Coventry, "Some effects of cognitive style on learning UNIX," *International Journal of Man-Machine Studies (1989)* vol. 31, pp. 349-365.

[8] P. T. Cox, F. R. Giles, and T. Pietrzykowski, "Prograph: a step towards liberating programming from textual conditioning," *1989 IEEE Workshop on Visual Languages*, Oct. 4-6, 1989, Rome, Italy, pp. 150-156.

[9] P. T. Cox and T. Pietrzykowski, "Using A Pictorial Representation to Combine Dataflow and Object-Orientation in a Language Independent Programming Mechanism," *Proceeding International Computer Science Conference 1988*, pp. 695-704.

[10] J. J. Craig, *Introduction to Robotics*, 2nd Ed. (Reading, Massachusetts:Addison Wesley Publiching Company), ©1989.

[11] J. F. DeSoi, W. M. Lively, and S. V. Sheppard, "Graphical Specification of User Interfaces with Behavior Abstraction," *Proceedings of CHI '89*, pp. 139-144.

[12] P. R. Frey, W. B. Rouse, and R. D. Garris, "Big Graphics and Little Screens: Designing Graphical Displays for Maintenance Tasks," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 1, Jan./Feb. 1992.

[13] D. R. Gentner and J. Grudin, "Why Good Engineers (Sometimes) Create Bad Interfaces," *CHI '90 Proceedings*, pp. 277-282.

[14] M. W. Gertz, "Simulation of Strategies to Minimize the Force of Impact between a Redundant Manipulator and an Obstacle," M.S. Thesis, Department of Elec. and Comp. Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, February 1991.

[15] M. W. Gertz, D. B. Stewart, and P. K. Khosla, "An Iconic Programming Language for Sensor-Based Robots," *Proceedings of SOAR Conference*, Aug. 4-6, 1992, Houston, Texas.

[16] M. W. Gertz, D. B. Stewart, and P. K. Khosla, "A Software Architecture-Based Human-Machine Interface for Reconfigurable Sensor-Based Control Systems," *Proceedings of 8th IEEE International Symposium on Intelligent Control*, Aug. 25-26, 1993, Chicago, Ill., pp.75-80.

[17] M. W. Gertz, D. B. Stewart, and P. K. Khosla, "A Human-Machine Interface for Reconfigurable Sensor-Based Control Systems," *Proceedings of Conference on Space Programs and Technologies: AIAA*, Sept. 21-23, 1993, Huntsville, Alabama.

[18] M. W. Gertz, D. B. Stewart, and P. K. Khosla, "Onika: A Human-Machine Interface for Reconfigurable Sensor-Based Control Systems," *IEEE Robotics and Automation Magazine* (accepted; to be published March 1995).

[19] M. W. Gertz and P. K. Khosla, "Onika: A Multilevel Human-Machine Interface for Real-Time Sensor-Based Systems," *ACSE: Proceedings of SPACE '94, March 1994*, Albuquerque, New Mexico.

[20] M. W. Gertz, D. B. Stewart, B. J. Nelson, and P. K. Khosla, "Using Hypermedia and Reconfigurable Software Assembly to Support Virtual Laboratories and Factories," *Proceedings of 5th International Symposium on Robotics and Manufacturing,* Aug. 15-17, 1994, Maui, Hawaii, vol. 5, pp. 493-500.

[21] M. W. Gertz and P. K. Khosla, "Visual Programming and Program Visualization Techniques for Port-Based Objects," *Proceedings of 1994 Annual Meeting of the American Nuclear Society*, June 19-23, 1994, New Orleans, Louisiana.

[22] M. W. Gertz and P. K. Khosla, *The Onika 1.3 User's Manual*, Program Documentation, Dept. of Elec. and Comp. Engineering and the Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213, July 1994.

[23] E. P. Glinert, "Out of Flatland: Towards 3-D Visual Programming," *IEEE Proceedings 2nd Fall Joint Computer Conference 1987*, pp 292-299.

[24] E. P. Glinert and S. L. Tanimoto, "Pict: An Interactive Graphical Programming Environment," Computer, *November 1984*, pp. 7-25.

[25] E. J. Golin and S. P. Reiss, "The Specification of Visual Language Syntax," *IEEE Proceedings Workshop on Visual Languages 1989*, pp. 105-110.

[26] K. Gronbaek and R. H. Trigg, "Design Issues for a Dexter-Based Hypermedia System," *Communications of the ACM*, vol. 37, no. 2, February 1994, pp. 40-49.

[27] F. Halasz and M. Schwartz, "The Dexter Hypertext Reference Model: Hypermedia," *Communications of the ACM*, vol. 37, no. 2, February 1994, pp. 30-39.

[28] K. H. Hanne and J. Ph. Hoepelman, "Combined Graphic- and Natural Language-Interaction (Design and Implementation)," *Proceedings of Graphics Interface '88*, June 6-8, 1988, Edmonton, Alberta, pp. 105-111.

[29] R. W. Harrigan, "Automating the Control of Robotics Systems in Unstructured Environments," *Proceedings of 8th IEEE International Symposium on Intelligent Control*, Aug. 25-26, 1993, Chicago, Ill., pp. 81-89.

[30] D. Heller, *XView Programming Manual, Third Edition, for XView Version 3*, ©1991 O'Reilly & Associates, Inc., Sebastopol, California.

[31] R. Henneman and W. Rouse, "Measures of Human Problem Solving Performance in Fault Diagnosis Tasks," *IEEE Transactions on Systems, Man, and Cybernetics*, January/February 1994, pp. 99-112.

[32] M. Hirakawa, M. Yoshimi, M. Tanaka, and T. Ishikawa, " A Generic Model for Constructing Visual Programming Systems," *1989 IEEE Workshop on Visual Languages*, Oct. 4-6, 1989, Rome, Italy, pp. 124-129

[33] W. Horton, "Overcoming Chromophobia: A Guide to the Confident and Appropriate Use of Color," *EEE Transactions on Professional Communication*, vol. 34, no. 3, September 1991, pp. 160-173.

[34] S. Hudson, "Adaptive Semantic Snapping — A Technique for Semantic Feedback at the Lexical Level," *CHI '90 Proceedings*, pp. 65-69.

[35] *Inside Macintosh Volume VI*, ©1991 Apple Computer, Inc.

[36] M. E. Kopache and E. P. Glinert, " C^2: A Mixed Textual/Graphical Environment for C," *IEEE Proceedings Workshop on Visual Lanaguages 1988*, pp. 231-238.

[37] L. Leifer, M. Van der Loos, and D. Lees, "Visual Language Programming: for Robot Command-Control in Unstructured Environments," *Proceedings of the 5th International Conference on Advanced Robotics: Robots in Unstructured Environments*, June 19-22, 1991, Pisa, Italy, pp. 31-36.

[38] K. Lodding, "Iconic Interfacing," *IEEE Computer Graphics and Applications*, vol. 3, no. 2, March/April 1983, pp. 11-20.

[39] *Macintosh User's Guide for Desktop Macintosh Computers*, ©1992 Apple Computer, Inc.

[40] A. MacLean, K. Carter, L. Lövstrand, and T. Moran, "User-Tailorable Systems: Pressing the Issues with Buttons," *CHI '90 Proceedings*, pp. 175-182.

[41] D. E. Mahling and D. Fisher, "The Cognitive Enginering of Visual Languages," *1990 IEEE Workshop on Visual Languages*, Oct. 4-6, 1990, Skokie, Illinois, pp. 22-27.

[42] D. E. Mahling, O. A. Sandvik, and W. B. Croft, "Visual Interaction between End Users and Goal Based Systems," *1990 IEEE Workshop on Visual Languages*, Oct. 4-6, 1990, Skokie, Illinois, pp. 182-186.

[43] T. Malone and M. R. Lepper, "Making Learning Fun: A Taxonomy of Intrinsic Motivations for Learning," *Aptitude, Learning, and Instruction, Volume 3: Conative and Affective Process Analyses*, R. E. Snow and M. J. Farr (ed.), ©1987 Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey, pp. 223-253.

[44] R. A. Maxion, "Toward Fault-Tolerant User Interfaces," *Proceedings of the 5th IFAC International Conference on Achieving Safe Real-Time Computing Systems (SAFECOMP-86)*, Sariat, France, October 1986.

[45] D. Meister, "System Effectiveness Testing," *Handbook of Human Factors*, pp. 1271-1297 (New York: John Wiley & Sons, Inc.; G. Salvendy, ed.), ©1987.

[46] W. Meyer (principal investigator), "Final Report, NASA Contract NAS1-18393," Grumman Data Systems, Technology Department, June 27, 1988, pp. 34-47.

[47] F. Modugno and B. Myers, "Pursuit: Visual Programming in a Visual Domain," Tech Report CMU-CS-94-109, School of Computer Science, Carnegie Mellon University.

[48] G. M. Murch, "Physiological Principles for the Effective Use of Color," *IEEE Computer Graphics and Applications*, vol. 4, no. 11, November 1984, pp. 49-54.

[49] P. Mussio, M. Pietrogrande, M. Protti, F. Columbo, M. Finadri, and P. Gentini, "Visual Programming in a Visual Environment for Liver Simulation Studies," *1990 IEEE Workshop on Visual Languages*, Oct. 4-6, 1990, Skokie, Illinois, pp. 29-35.

[50] B. A. Myers, "Taxonomies of Visual Programming and Program Visualization," *Journal of Visual Languages and Computing*, 1990 (1), pp. 97-123.

[51] The NCSA Mosaic Home Page, on-line documentation, http://www.ncsa.uiuc.edu/SGD/Software/Mosaic/NCSAMosaicHome.html, ©1993 University of Illinois, Urbane-Champaign, Illinois.

[52] A. Nye (ed.), *Xlib Programming Manual for Version 11 of the X WIndow System* (vol. 1), ©1990 O'Reilly & Associates, Inc., Sebastopol, California.

[53] A. Nye (ed.), *Xlib Reference Manual for Version 11 of the X WIndow System* (vol.2), ©1990 O'Reilly & Associates, Inc., Sebastopol, California.

[54] K. Potosnak, "Do Icons Make User Interfaces Easier to Use?" *IEEE Software*, May 1988, pp. 97-99.

[55] *ResEdit™ Reference for ResEdit version 2.1*, ©1991 Apple Computer, Inc.

[56] D. L. Schneider, "Review of the Robotica Software Package for Robotic Manipulators," *IEEE Robotics and Automation Magazine*, vol. 1, no. 1, March 1994, pp. 21-22.

[57] D. B. Stewart, "Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based Systems," Ph.D. Thesis, Carnegie Mellon University, April 1994.

[58] D. B. Stewart, M. W. Gertz, and P. K. Khosla, "Software Assembly for Real-Time Applications Based on a Distributed Shared Memory Model," in *Proceedings of the 1994 Complex Systems Engineering Synthesis and Assessment Technology Workshop (CSESAW '94)*, Silver Spring, Maryland, July 1994, pp. 214-224.

[59] D. B. Stewart and P. K. Khosla, *Chimera 3.1 Real-Time Programming Environment*, Program Documentation, Dept. of Elec. and Comp. Engineering and the Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213, July 1993.

[60] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Design of Dynamically Reconfigurable Real-Time Software using Port-Based Objects," The Robotics Institute, Carnegie Mellon University Tech. Report #CMU-TR-RI-93-11, July 1993.

[61] S. L. Tanimoto and E. P. Glinert, "Designing Iconic Programming Systems: Representation and Learnability," *IEEE Proceedings Workshop on Visual Languages 1986*, pp. 54-60.

[62] S. L. Tanimoto and M. S. Runyan, "PLAY: An Iconic Programming System for Children," *Visual Languages*, S. K. Chang, T. Ichikawa, and P. A. Ligomenides (ed.), pp. 191-205, ©1986 Plenum Publishing Corporation.

[63] User's Guide to VAL II, Preliminary Program Documentation, ver. X2, April 1983, ©1983 Unimation, Inc.

[64] T. Van Raalte (ed.), *XView Reference Manual for XView Version 3*, ©1991 O'Reilly & Associates, Inc., Sebastopol, California.

[65] H. A. Witkin and D. R. Goodenough, *Cognitive Styles: Essence and Origins: Field Dependence and FIeld Independence*, Psychological Issues, Monograph 51, ©1981 International Universites Press, Inc., New York.

[66] H. T. Wood and S. K. Wood, "Icons in Everyday Life," *Human-Computer Interaction — INTERACT '87*, H.-J. Bullinger & B. Shakel (ed.), Elsevier Science Publishers B.V. (North Holland), ©IFIP, 1987.