# An Iconic Programming Language for Sensor-Based Robots

Matthew Gertz, David B. Stewart, Pradeep K. Khosla

Department of Electrical and Computer Engineering
The Robotics Institute
Carnegie Mellon University, Pittsburgh, PA. 15213

*Abstract*— **In this paper we describe an iconic programming language called Onika for sensor-based robotic systems. Onika is both modular and reconfigurable and can be used with any system architecture and real-time operating system. Onika is also a multi-level programming environment wherein tasks are built by connecting a series of icons which in turn can be defined in terms of other icons at the lower levels. Expert users are also allowed to use control block form to define servo tasks. The icons in Onika are both shape and color coded like pieces of a jigsaw puzzle thus providing a form of error control in the development of high level applications.**

**Keywords- Programming, Real-time, Iconic Language, Sensor-based robots, visual programming.**

## 1 Introduction

Programming manipulators to perform tasks can often be a difficult and frustrating task. Many of the programming languages available have a C-like syntax, which makes developing applications very difficult for persons not having an adequate background in programming. Deciphering and debugging cryptic, non-portable, and ill-commented code can waste many man-hours of valuable time, while executing such questionable code can be physically dangerous to both machine and operator. Man-hours are also wasted when an operator must undergo lengthy training to be able to operate a robotic system. While C-language programming is appropriate for experienced programmers, it is inappropriate for users who are interested only in effectively using a manipulator or a robotic system. What is, therefore, needed is a programming language and a system that simultaneously provides the ability to develop systems level code and also allows users to program applications easily.

At Carnegie Mellon University, we are pursuing research with the goal of creating a programming and control environment, for sensor-based systems, that allows for rapid development of applications through automatic generation and validation of real-time code. In order to make sensor-based robot systems easy to use and program, we are also developing an *iconic programming language* (IPL) called *Onika* for use as a *human-machine interface* for programming robotic systems. In this paper, we provide an overview of the current version of Onika and its capabilities for programming sensor-based systems.

Onika is a multi-level programming environment that is both modular and reconfigurable. At the highest level applications for a sensor-based manipulator[1] are created by choosing icons which represent objects, jobs, and expressions, and arranging them in a logical sequence. At lower levels, robotics-savvy users (or experienced programmers) can additionally define jobs, new icons, etc. for inclusion into the applications. At the lower level, it is also possible to combine icons representing tasks into control-block form and to bind C-language code to an icon. Once a task level iconic program is created in Onika, the underlying system provides the capability to develop the equivalent C-program for execution. The underlying system consists of the reconfigurable software structure[10] and the Chimera real-time operating system that we have developed [3].

In contrast to the previous work done on VPLs, described in the next section, we are developing our IPL to be reconfigurable, customizable, and able to sit in any architecture, drawing upon the resources of the resident real-time operating system (such as OS-9, CHIMERA II, VxWorks, etc.). The icons in Onika are both shape and color coded and can be thought of as pieces of a jig-saw puzzle. This is advantageous because a novice user cannot connect completely arbitrary icons to develop a program, at the task level, that does not make logical sense.

Onika also permits smaller sets of iconic programs to be combined to create larger programs (represented by one icon), thus allowing for the rapid creation of a library of tried-and-true procedures for the manipulator. Furthermore, by loading in the specifications of any particular manipulator, it is planned that our IPL will be able to run its programs on different manipulator systems *without the user needing to rewrite the code.* Translation would be done at a lower level in the system architecture, with the help of a *specs file* created for each manipulator (which would list construction data, joint limits and lengths, moments of inertia, and so forth). Finally, development of a lower-level simulator for the IPL is underway, so that an application created for a manipulator can be simulated in real-time without changing the application at all -- from a telerobotic standpoint, there would be no difference in the type of information received and the amount of control exerted in a simulation as opposed to an actual run.

1. Throughout this paper, we shall use the term *manipulator* to refer to the system programmed by the IPL; nevertheless, it should be noted that this IPL, being reconfigurable and modular, could be used for a variety of systems, such as satellite control, deep-sea remote exploration, or industrial process control.

If the use of manipulators is ever to become more widespread than it currently is, then the ability to program these machines must become available to the researcher or worker who may not have a background knowledge in computers or robotics. By introducing an IPL into existing manipulator systems, floor-level workers will be able to run complex and critical routines, while the actual coding of lower-level tasks for these machines that the icons represent can be limited to professional programmers.

This paper is organized as follows: The next Section describes previous efforts in this area and in Section 3, we briefly discusses the history of our IPL, and how we implemented a test version to demonstrate the effectiveness of such a scheme. Section 4 describes our current version of IPL called Onika and in Section 5, we discuss several research issues that we are pursuing. Finally, in Section 6, we summarize the paper.

## 2 Previous Work

An iconic programming language is distinct from what is known as a *visual programming language* (VPL), in that an IPL[1] relies on the user's association of actions and objects with pictures, shapes, and colors, rather than with less-informative flowcharts, textual information, and cryptic coding sequences. Iconic interfaces can be very useful for training new users to effectively operate their systems. Icons have proven to be easily identifiable and have simplified the tasks of moving through directories, accessing files, and running executables as in Macintosh and Windows environments. An IPL extends this idea by identifying an icon with a specific action or object; by combining icons in a certain way, an application can be described and executed.

A considerable amount of effort has been expended for developing graphical interfaces and visual languages to define an application. Visual programming has been applied to many diverse domains and an excellent review may be found in [11] . In this short review of previous work, our goal is to provide a perspective and motivation for the features that we chose to include in the development of Onika.

Hanne and Hoepelman suggest a "natural language" (NL) approach, where questions and statements are made based on simple English sentences[2] . The disadvantage of this is that "natural language" is only natural to those who communicate in an Indo-European language. For instance, the structure of Chinese and Japanese languages are quite different than that of English, and Amelsan (American Sign Language) structurally bears little resemblance to any spoken language. We feel that to be useful, an IPL should be conceptual, rather than English-oriented, and we have used this approach in our work. There are precedents for this approach; for instance, international traffic signs use concepts rather than written language. Besides, a visual robotic programming language using the NL approach would be so complex to parse as to minimize any advantage gained from using icons.

Leifer et. al. use icons to represent manipulation primitives [4]; these icons are then combined to construct a manipulation program. For the non-engineering type of person, programs can be easily created, debugged, and modified without a detailed knowledge of programming or computers in general. However, for the engineer who must decide whether or not to use PID control, force control, hybrid control, and so forth, the lexicon of the language of the interface must be expanded to include more than one level of programming. Furthermore, conditionals (such as if/then/else) should be added to create a robust vocabulary, and a grammar must be added so that the user is kept from creating impossible routines (e.g. *Move-To Move-To Ball Move-To Pick-Up*). Furthermore, the IPL should allow the creation of parallel-task applications, and the user should be able to completely redefine icons and their meanings without diminishing portability. In our research we are developing methods to implement these necessary programming concerns in a visual manner which is easily understood by the user.

Mahling and Croft introduce a visual programming language for the acquisition and display of plans for a planning system [5]. They have icons for acts and icons for relations and objects, which are used for creating plans to achieve some logical goal. What is noteworthy about their set-up is the notion of an expandable library of icons. Clearly, this is a beneficial thing to have. However, to minimize user confusion, only appropriate icons should be available to the user – for example, if a manipulator doesn't have a camera, then vision-routine icons should not available as icon choices for an application during its creation. Some method of organizing icons by category and domain is needed to prevent users from creating grammatically-correct but non-functional applications. A method of categorization of types and purposes of icons into some grammatical dichotomy is a major goal of our research.

Flow chart methods are often suggested, as Angelaccio et. al. have done for E-R oriented databases [1]. While flow charts lend themselves well to mapping the flow of a routine, they can look intimidating and are not as friendly to the user in terms of presentation of information as pictures would be. This is a much more important consideration than many people might think; user-friendliness improves productivity. Furthermore, arrows from one flow element to the next unnecessarily waste screen space. In the iconic approach we propose the creation of applications from job icons, the icons sit adjacent to each other; since icons do not require text to describe them, they can be much smaller than flow chart elements and yet convey the same amount of information and sense of program direction. This minimizes wasted space, and allows more routine elements to be seen at a time, which aids application development.

Flow chart methods are not at all useful when describing how a job is defined by servo tasks running simultaneously, and so

1. Although *IPL* technically refers to the grammar, syntax, and manipulation of the programming elements of a particular type of human-machine interface, rather than the interface to the machine itself, we shall use both the terms *IPL* and Onika throughout this paper as reference to both the language and the interface.

when defining jobs from task we use a control-block form. Additionally, the pictures in the icons are more easily identifiable than the generic boxes of a flowchart with respect to specific routines. A flowchart element must represent a "complete" event (action + objects), since all flowchart elements of a type should theoretically be interchangeable. An IPL, on the other hand, could have an object easily replaced in an event without effecting its dependent action (and all of the values possibly associated with it), and vice-versa.

Mussio et. al. use icons which vary in shape, and are thus assembled in sort of a jigsaw puzzle ensuring correct grammar [7]; certain icons can only interlock with other icons (both left-right and up-down). Their icons resembled liver cells; a hepatologist was guided to create valid models of the liver and test them by using it. Glinert [12] describes a system called Proc-BLOX that uses jigsaw puzzle pieces to present program constructs. Onika uses icons, as mentioned before, that are both shape and color coded puzzle pieces. In this context, a foremost concern in our research is exploring the best way for icons to be identified by class and grammar, whether by shape, color, size, or some other visual difference.

In the next section, we describe an initial version of our IPL called *Bookworm*.

## 3 A History of Our IPL

Our first prototype IPL called Bookworm was developed to demonstrate the effectiveness of iconic programming, and to discover some of the issues which would demand investigation in the research and development of a more powerful IPL. The Bookworm IPL was used to combine jobs and objects into applications, where the jobs were defined using code (in the newer IPL, Onika, the jobs used in the creation of applications are themselves iconically defined rather than textually defined).

Bookworm used a specific grammar to decide which icons may follow other icons in a story. For instance, an action icon which required an object (e.g. *Move-To* <object>) could not be followed by another action icon; it had to be followed by an object. Similarly, objects had to be preceded by an appropriate action-requiring-object icon. The grammar was immediately understandable to the user, since we used colors to identify icons: an icon whose right half is blue must be followed by an icon whose left half is blue, and so forth. There were three different types of icon "words" (green-green, green-blue, and blue-green, representing *action*, *action requiring object*, and *object*, respectively; we are currently performing research to discover exactly how many different classes of words are required for our latest IPL, Onika). The grammar of an icon was defined when the icon was created or modified, so that color did not need to be the identifying key; it could instead have been the shapes of the edges of the icons, for instance (useful if the user is color-blind).

Bookworm also conformed to the conventions of the platform on which it exists. For instance, the Macintosh version could be run from pull-down menus and command keys as well as icons; the Sun version of our present IPL, Onika, similarly uses SunView and XView conventions. In this way, users familiar with a particular platform are more easily be able to anticipate how the IPL reacts to commands and keystrokes.

Although Bookworm did not run a program on an actual manipulator, it did generate simulation files for use with ROBSIM, a NASA/Langley robotic simulator. When the "Simulate Story" icon was selected, AL code was generated through a four-pass method. First, Bookworm looked for holes in the story, and aborted the simulation if it found any. Next, Bookworm checked value panels for object locations, and defined variables for those locations, which it inserted into the AL file. On the next pass, those variables were initialized, and, finally, the meat of the code was produced.

Bookworm was strictly a higher-level IPL. At an early stage, we recognized that lower-level routines could also be coded using icons, although an entirely different grammar would be required, since lower-level routines are very specific and represent quite abstract concepts. Onika, the current IPL under development, is much more stratified than Bookworm and can be used to create lower-level jobs as well as higher-level applications, making itself more useful to programmers while still keeping lower-level details transparent to less technically-oriented users.

Onika's higher-level interface under development resembles Bookworm strongly but allows for a much expanded grammar (including conditionals, loops, and error-trap routines). By using icons and visual grammar cues to identify procedures, we avoid many of the problems of the flowchart approach to visual programming, including usage of screen space (much of which is wasted in flowchart methods), dependence on English, and problems in recognition of routines (one flow chart box looks pretty much like any other). Our icon "stories" are concise, compact, and easily read.

Onika's lower-level interface, designed to be used by experienced persons, resembles nothing so much as an ICCAD interface. Icons representing tasks, and having certain input and output pins, are combined into control block diagrams (connections are done automatically by the system, relieving the user of that tedious burden). Instead of having to change lines in pages of cryptic real-time code, the user can manipulate tasks graphically, retrieving and changing task information simply by clicking on the task's icon with a mouse. Activation and deactivation of one or all tasks is done with one keystroke, and task configurations can be modified as simply as selecting a task, deleting it, and dragging over another task to take its place.

Clearly, both the lower- and higher-level interfaces of Onika must rely on an underlying software support structure. Onika can be tailored for use on any system architecture, using any real-time system. The following section discusses Onika and its relationship to our own system architecture.

## 4 Onika's Position in the System Architecture

Figure 1 shows the system software architecture for reconfigurable sensor-based control systems[9][10] that we have developed. In this architecture, we have defined several types of routines. The most general manipulator routine is the *application*, which specifies some goal to be achieved. For example, "Wash the dishes" is a fairly typical application in day-to-day life. An application is defined by some serial flow (or flows) of *jobs*, which define some singular action affecting an object. "Lift the plate," "Put scrubber on plate," and "Scrub" are good examples of jobs executed serially to achieve the application goal of washing the dishes. Finally, a job can be loosely described as being defined by a collection of *tasks*, all of which are running concurrently to fulfill the conditions of a job, and all of which have certain input and output functions. "Perform inverse dynamics," "Resolve acceleration of scrubbing unit," and "Invert scrubbing-arm jacobian" are all examples of tasks running concurrently to finish the job "Scrub." Tasks themselves are defined by various subroutine calls and are textually coded in *C* language.

Onika can be used to create both jobs from tasks *and* applications from jobs. Users can open a *task lexicon* (Figure 2), and drag icons representing tasks from the lexicon to a *job canvas* (these icons are simple CAD representations, and are created on the fly). Once on the canvas, a task is created on the underlying real-time system, and the icon representing that task automatically connects itself to other task icons based on the input/output variables of all the tasks in question (Figure 3). This allows the user to easily see whether or not a complete job has been created. The user can modify certain values associated with each task, such as the frequency, the priority, and whether or not the task is active or inactive. Task icons can be cut and repasted, although only one instance of each task can exist on the canvas. To create a job from tasks, the user need not know anything about the underlying real-time system, nor know much about computers, but he or she must have some knowledge about robotics.

However, once a job has been completely defined (shown in Figure 4), the entire job can be saved and linked to an pictorial icon (Figure 5). This pictorial icon can be saved for general use to a *job dictionary*, and dragged to an *application workspace* to become part of an application (Figure 6), which can also be saved and reloaded. When the application is run, and the job is encountered in a program flow, the tasks associated with that job are automatically loaded and activated based on the manipulator description chosen by the user. When the job is complete, its tasks are destroyed, and the program flow moves to the next job in the application. Jobs can cut, copied and pasted to the application as well, and of course multiple instances of jobs in the application are allowed. All of this is transparent to the high-level user. In order to create applications from jobs, the user need not know anything about the underlying support system, computers, or even anything about robotics. He or she simply needs to know how to drag job icons from one location of the screen to the other in some meaningful serial order ("Move here, then do this, then move there, then do that..."), the background grammar-checker ensuring that syntactically impossible applications cannot be generated.

Although not yet implemented, it is planned that applications will be used to define other applications (for instance, the application "Wash the dishes" could be a sub-application of the application "Do the housework"), and that applications could follow several flow branches (or even parallel flow branches) based on conditions at some point in the application.

It is expected that non-technically-oriented users would primarily create applications from jobs, and that the more robot/computer-oriented user would create the jobs from tasks. Experiments with the prototype IPL Bookworm have shown that most users can learn create usable applications from jobs in less than a half-hour.

While the implementation of an interface for technically-oriented users to define jobs from tasks is fairly straightforward, the implementation of powerful yet user-friendly interface for people having little or no background in robotics or computers to create applications from jobs is not. The next section discusses some of the research issues that we will be exploring while developing the IPL and supporting architecture.

## 5 Research and Development Issues

In order to develop the iconic human-machine interface for creating *applications* from *jobs*, several important research and development issues will need to be explored. These include (but are not limited to) the development of a grammar and the presentation of information to the user.

First, a grammar for the interface must be developed. This task, in turn, raises three other issues: first, how to represent a concept graphically and identifiably within a limited amount of screen space (the icon); second, how to classify the job icons as to grammatical types and identifying the number of types that will be needed, such as "self-contained action," "action requiring object," "object," "conditional," "modifier," etc. (the dichotomy); and third, how present visual grammatical clues to the user, so that the he or she does not waste time by attempting to create non-grammatical applications (the syntax).

In addition to the development of a grammar, the organization of information on the screen, and the devices by which it is affected, are also of paramount importance, and research will be performed to maximize their effectiveness. Interface controls to create, simulate, and run manipulator applications must be easily interpretable and used. The presentation of the application under development must allow the user to clearly see and understand the routine he or she is creating. Background error checking should immediately indicate when an impossible application is being built and prevent its occurrence, so that later debugging is kept to a minimum. The interface itself should conform to the established customs of the platform on which it exists (for instance, on the Macintosh, one would expect pull-down menus and command keys to perform operations similar to those that the user would expect they would in other Macintosh programs). Finally, all lower-level details must be transparent to the user; the interface should be developed with non-technically-oriented users in mind. These steps are necessary to create an interface which can be used at all times, and under any conditions, by users at any level of technical expertise.

We are especially interested in determining if an iconic programming interface will reduce the training time, experience, and education necessary to operate machinery which at lower levels is quite complex. Informal tests have shown that users who have some familiarity with computers and window interfaces can learn to proficiently use the prototype IPL Bookworm in less than a half-hour. We plan on doing extensive tests on an expanded version to see how users who are inexperienced with using computers and/or robotics adapt to Onika.

## 6 Summary

We have described an iconic programming interface for creating applications for sensor-based systems such as manipulators. This IPL, called Onika, will allow a user to control both higher-level and lower-level routines. Lower-level details are kept transparent to non-technically-oriented users. With only a couple of hours of training, we expect that users will be able to construct complex applications for manipulators, despite any lack of previous experience with programming and/or robotics. To make the IPL as effective as possible, research is being performed to determine the proper grammar and visual presentation for the IPL.

## Acknowledgments

Partial support for this research was provided by NASA, Sandia National Laboratories, the Department of Electrical and Computer Engineering, and The Robotics Institute, Carnegie Mellon University.

The authors would like to thank Philip Morris, Mike Goode, [illegible] Pennington, and all of the other researchers at NASA/Langley's LTM facility for the use of machines and time to complete the Bookworm prototype IPL. Additional thanks go to Dr. Rich Volpe at JPL for his help in defining the task/job relationship.

**More information** on Onika, Chimera real-time operating system, and reconfigurable software can be obtained by contacting Professor Pradeep K. Khosla at Department of Electrical and Computer, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213.

## References:

[1] Angelaccio, M., Catarci, T., and Santucci, G. "QBD*: A Fully Visual System for E-R Oriented Databases," 1989 IEEE Workshop on Visual Languages, Oct 4-6, 1989, pp. 56-61, Rome, Italy.

[2] Hanne, K. H. and Hoepelman, J. Ph. "Combined Graphic- and Natural Language-Interaction (Design and Implementation)," Proceedings of Graphics Interface '88, June 6-8, 1988, pp. 105-111, Edmonton, Alberta.

[3] D.B. Stewart, D.E. Schmitz, and P.K. Khosla, "Implementing real-time robotic systems using Chimera II," in *Proc. of IEEE Intl. Conf. on Robotics and Automation*, Cincinnati, OH, pp. 598-603, May 1990.Chimera

[4] Leifer, L., Van der Loos, M., and Lees, D. "Visual Language Programming: for robot command-control in unstructured environments," Proceedings of the Fifth International Conference on Advanced Robotics: Robots in Unstructured Environments, June 19-22, 1991, pp. 31-36, Pisa, Italy.

[5] Mahling, D. E. and Croft, W. B. "A Visual Language for the Acquisition and Display of Plans," 1989 IEEE Workshop on Visual Languages, Oct. 4-6, 1989, pp. 50-54, Rome, Italy.

[6] Miyao, J., Wakabayashi, S., Yoshida, N., and Ohtahara, Y. "Visualized and Modeless Programming Environment for Form Manipulation Language," 1989 IEEE Workshop on Visual Languages, Oct. 4-6, 1989, pp. 99-104, Rome, Italy.

[7] Mussio, P., Pietrogrande, M., Protti, M., Colombo, F., Finadri, M., and Gentini, P. "Visual Programming in a Visual Environment for Liver Simulation Studies," 1990 IEEE Workshop on Visual Languages, Oct. 4-6, 1990, pp. 29-35, Skokie, Illinois.

[8] Prusinkiewicz, P. and Knelsen, C. "Virtual Control Panels," Proceedings of Graphics Interface '88, June 6-8, 1988, pp. 185-191, Edmonton, Alberta.

[9] Stewart, D. B. "Real-Time Software Design and Analysis of Reconfigurable Advanced Sensor-Based Systems," Ph.D. prospectus, The Robotics Institute, Carnegie Mellon University, March 31, 1992.

[10] Stewart, D. B., Volpe, R. A., and Khosla, P. K. "Integration of software modules for reconfigurable sensor-based control systems," in *Proc. 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IROS '92), Raleigh, North Carolina, July 1992.

[11] Myers, B. A., Taxonomies of Visual Programming and Program Visualization, Journal of Visual Languages and Computing, 1990, pp 97-123.

[12] Glinert, E. P., Out of Flatland: Towards 3-D Visual Programming, Proceedings of 1987 Fall Joint Computer Conference, October, 1987, Dallas, TX. pp 292-299.

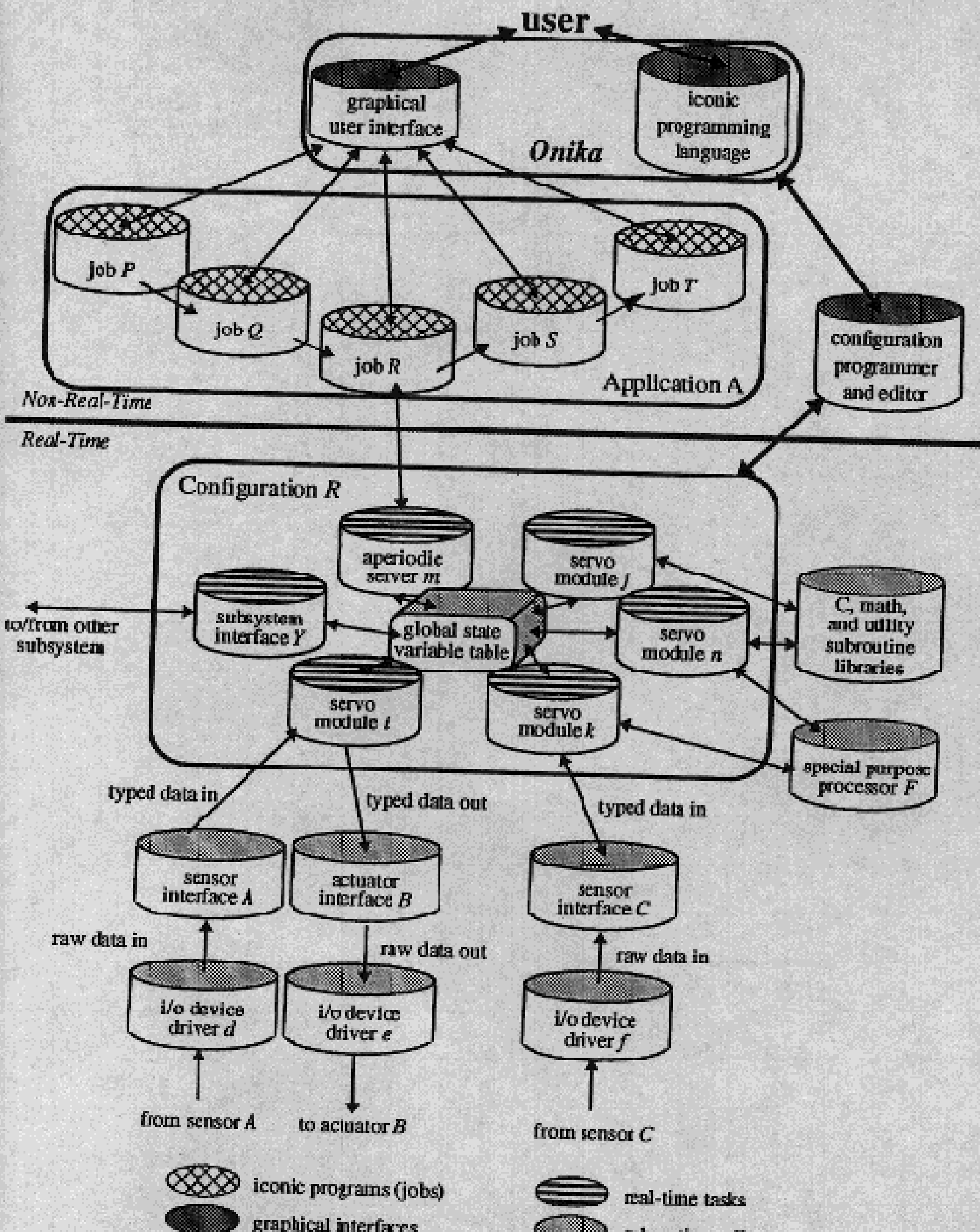


**Figure 1: Software Architecture for Reconfigurable Systems**

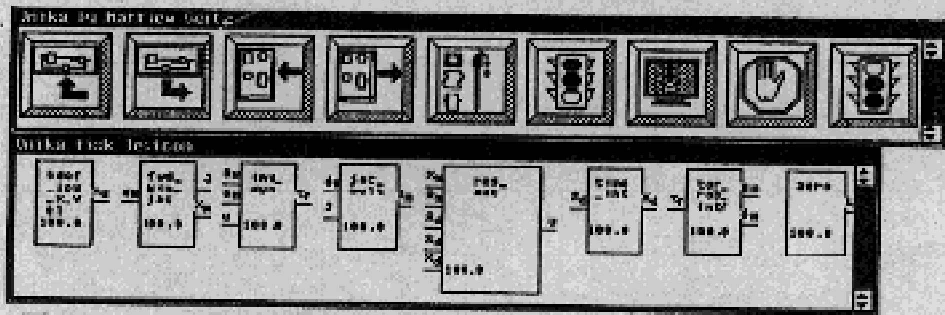

Figure 2: Loading the Task Lexicon into Onika (Sun version of Onika)

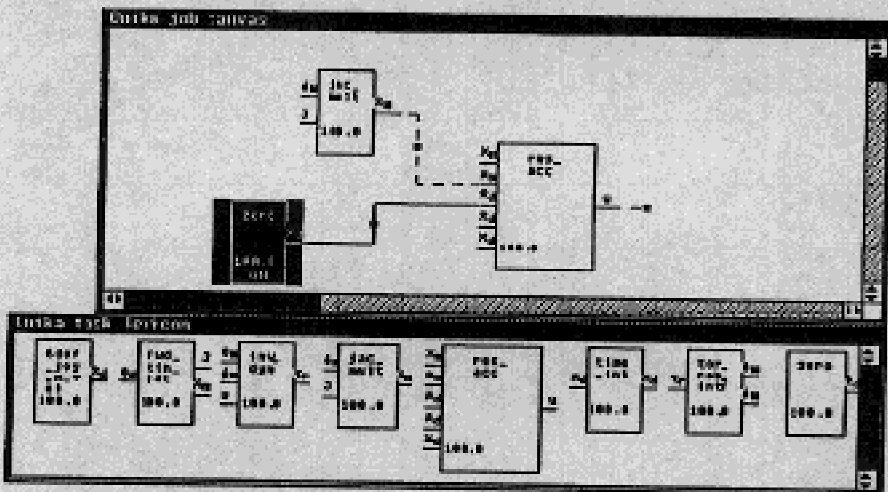

Figure 3: Pulling task icons onto the job canvas (Sun version of Onika)

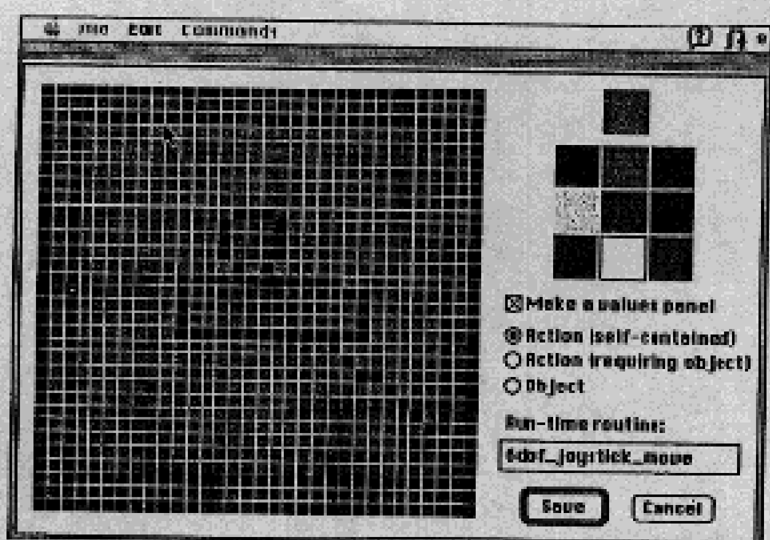


Figure 5: Creating an Icon for the Job (Mac II version of Bookworm)

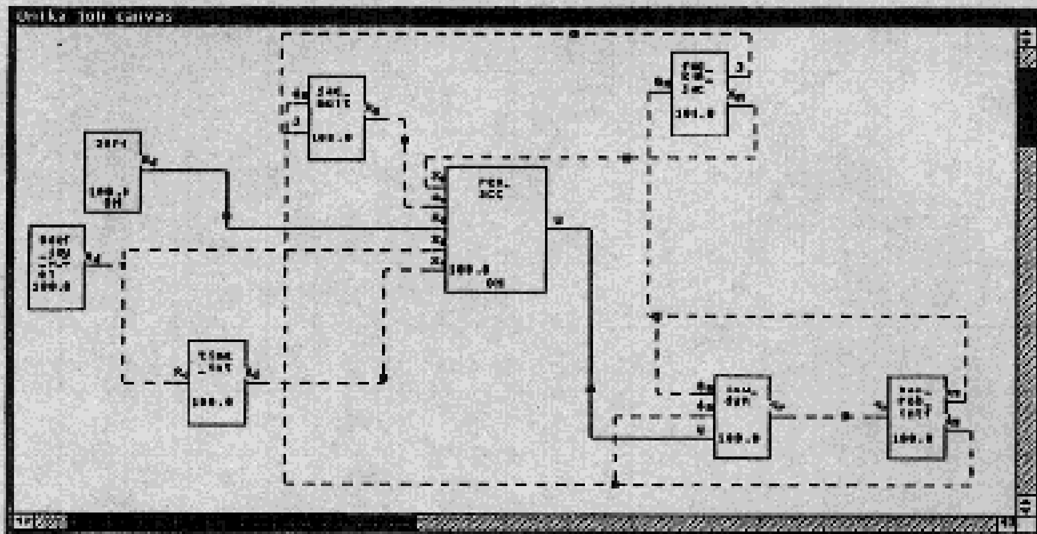


Figure 4: The completed job (Sun version of Onika). Note that two tasks are on and generating output as denoted by solid lines

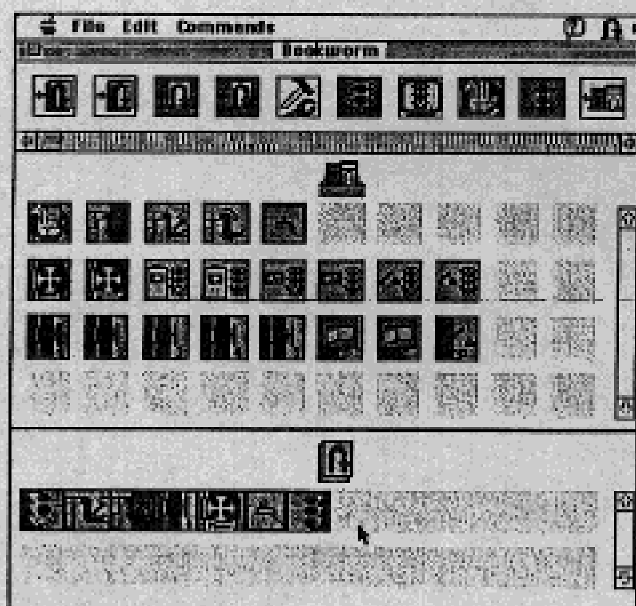


Figure 6: Using the Job's Icon in an Application (Mac II version of Bookworm)

SOAR '92 was published with incorrect figures. Here are the correct figures:

Correct figures for Gertz, Stewart, and Khosla, "An Iconic Programming Language for Sensor-Based Robots," SOAR 1992.

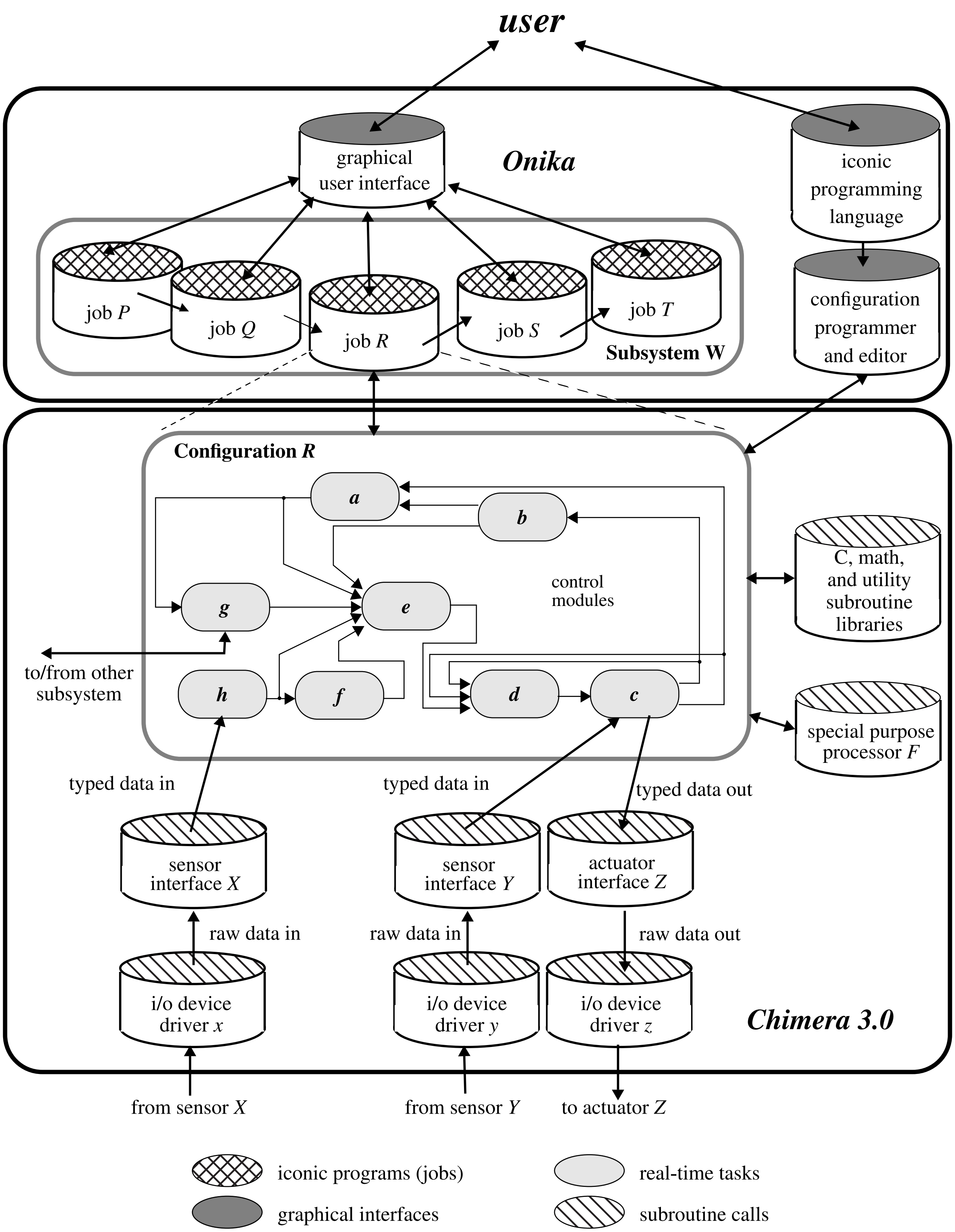


**Figure 1: Software Architecture for Reconfigurable Systems.**

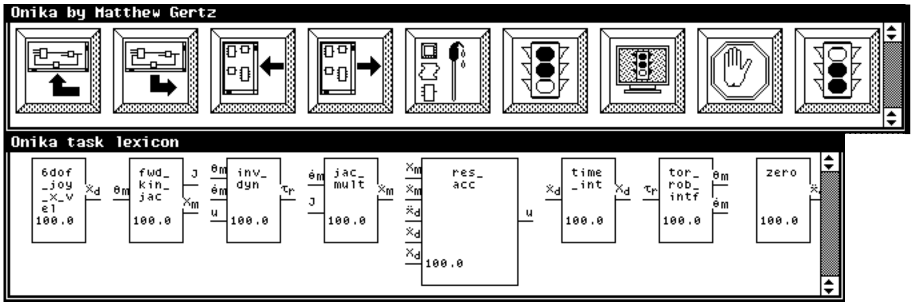


**Figure 2: Loading the Task Lexicon into Onika (Sun version of Onika). The lexicon contains directory references to various task files in the system; when the lexicon is loaded, these files are opened, and icons representing these tasks are created on the fly. These icons can be dragged to the Job Canvas, where they can be combined to define a job which the manipulator will perform.**

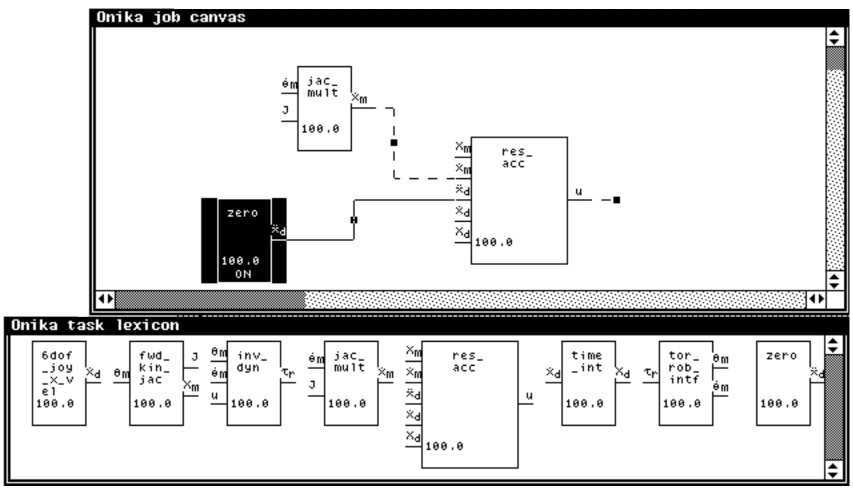


**Figure 3: Pulling Task Icons onto the Job Canvas (Sun version of Onika). The user has just dragged the icon "zero" to the job canvas, where it automatically connected itself to the icon representing the "resolved acceleration" task. Additionally, the user has turned the "zero" task on; it is generating output (a constant zero), and thus its connection line is solid. (Dashed lines, such as those coming from the "jacobian multiply" task and "resolved acceleration" task outputs, indicate that the task generating the output is not active.) The user can also select any icon in the lexicon or canvas with the middle mouse button to examine and change task values.**

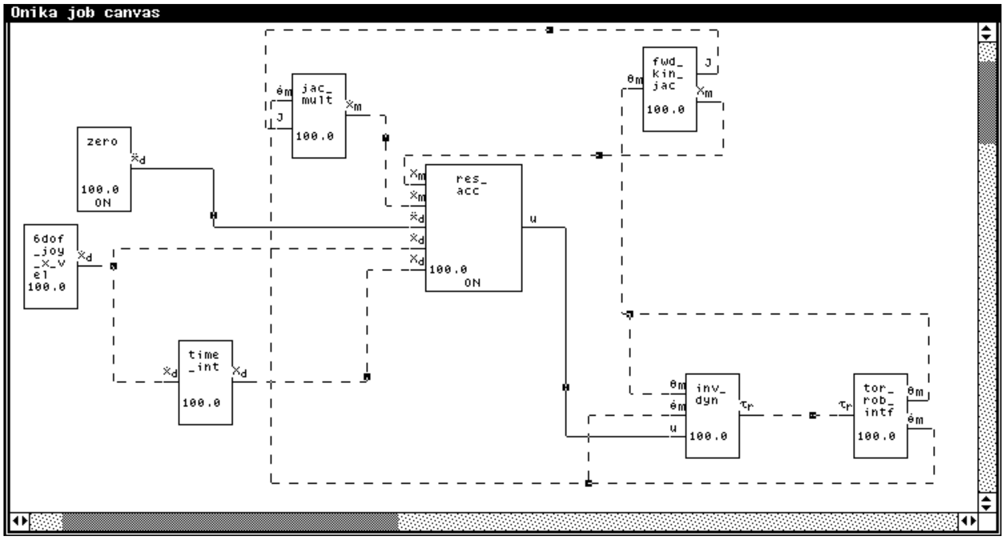


**Figure 4: The completed job (Sun version of Onika). Note that two tasks are on and generating output ("zero" and "resolved acceleration").**

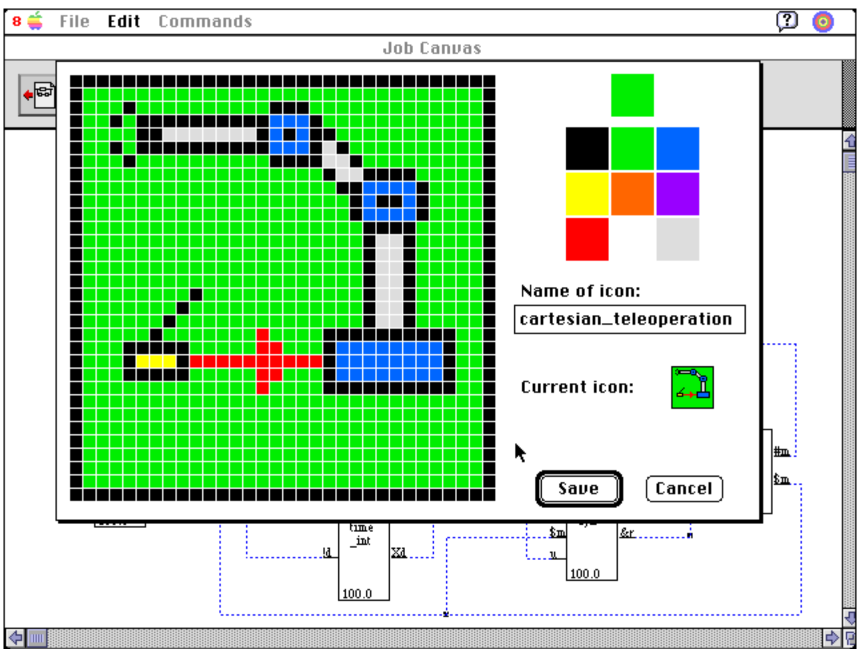


**Figure 5: Creating an Icon for a Job or Application (Mac II version of Onika). The user selects colors for the icon from a color palette and saves the icon. Onika will then determine the proper grammar type for the icon, whether it's an object, action, or other type of "word" being defined, and attaches the appropriate identifier shapes/colors on either side of the icon. The icon is identified with a certain existing application or job as chosen by the user, placed in a dictionary of the user's choice, and can thereafter be used in creating other applications. In this figure, the user is creating an icon to represent the job of Figure 4, so that the job may be included in a higher-level application.**

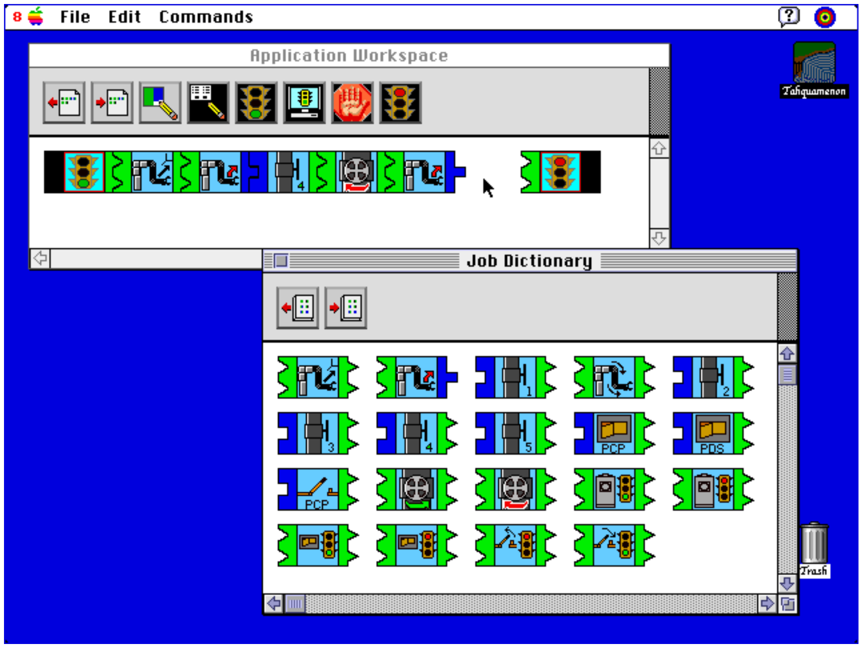


**Figure 6: Using a job's icon in an Application (Mac II version of Onika). The icons are dragged from the job dictionary and inserted into the application The application, if complete, can be run or simulated at any time. Additionally, by double-clicking on a job's icon, the user can examine and change values associated with that job. In this case, Onika inserted a space between the last two icons when the next-to-lact icon was inserted, since that icon requires an object to follow it in order for the application to be syntactically correct.**