Presented at "ASCE/SPACE 94: The 4th Int'l Conf. and Expo. on Engineering, Construction., and Operations in Space," Feb. 26-Mar. 3, 1994, Albuquerque, New Mexico.

Onika: A Multilevel Human-Machine Interface for Real-Time Sensor-Based Systems

Matthew W. Gertz[1] and Pradeep K. Khosla[2]

Abstract

The development of software for reconfigurable sensor-based real-time systems is a complicated and tedious process, requiring highly specialized skills in real-time systems programming. The total development time can be reduced by automatically integrating reusable software modules to create applications. The integration of these modules can be further simplified by the use of a high-level programming interface. We have developed Onika, an iconically programmed human-machine interface, to interact with a reconfigurable software framework to create reusable code. Onika presents appropriate work environments for both application engineers and end-users. For engineers, icons representing real-time software modules can be combined to form real-time jobs. For the end-user, icons representing these jobs are assembled by the user into applications. Onika verifies that all jobs and applications are syntactically correct, non-ambiguous, and complete. They can then be executed from within Onika, or can be saved as a stand-alone program which can be executed independently on the underlying real-time operating system. Onika has been fully integrated with the Chimera real-time operating system in order to control several different robotic systems in the Advanced Manipulators Laboratory at Carnegie Mellon University.

1. Introduction

The development of real-time software for *sensor-based 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. To do this, 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. In the past, researchers have created visual programming languages (VPLs) to address the problems of

[1] Ph.D. Graduate Researcher, Department of Electrical and Computer Engineering, The Robotics Institute at Carnegie Mellon University, Pittsburgh, PA 15213

[2] Associate Professor, Department of Electrical and Computer Engineering, The Robotics Institute at Carnegie Mellon University, Pittsburgh, PA 15213

textual coding [10][8][9][7][6][5][1]. 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 paper, we discuss the development of a multilevel/iconically-programmed human-machine interface called Onika. Onika has several abilities which increase its effectiveness with respect to other interfaces and programming environments for real-time sensor-based control systems. Onika directly connects with the underlying real-time operating system to coordinate the system's activities, giving a user a control capability which has not previously been available in interfaces for sensor-based systems. Programming can be done interactively or off-line. Onika gives the user access to a library of control modules, which are parallel-executing reusable software modules within a reconfigurable sensor-based control system. Each control module on the real-time operating system is represented by a block-form icon, which can be manipulated by a mouse. Using Onika, these icons can be 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 use Onika to monitor and modify the real-time performance and parameters of each routine running on the real-time operating system. Furthermore, a combination of routines created at one level of Onika can be saved as a reusable higher-level routine for others to use. Thus, routines at Onika's higher levels become more specific, making programming accessible for naïve users, without diminishing the programming scope for more knowledgeable users working at Onika's lower level. Unlike other interfaces, both levels of users, naïve and knowledgeable, are presented with an interface appropriate for their programming abilities and application requirements.

In section 2, we discuss various HMI/VPL systems which have been introduced in recent years. In section 3, we discuss the software framework in which Onika operates. In section 4, we introduce Onika, a multilevel iconic programming language (IPL) and human machine interface (HMI). We conclude this paper in section 5.

2. Previous Work

The problems associated with textual programming have been addressed on several levels in the past. (Comprehensive reviews of the state of visual programming techniques can be found in [10] and [1].) Researchers have created interfaces wherein routines for an existing programming language (such as C) are created by a higher-level VPL[10][6][5]. Interfaces such as these are designed to be used by programmers with knowledge of the structured programming language in question. They are best used for routines of lower- to middle-level rank. Higher-level HMIs have also been created for naïve users[10][8][9][7][1]; however, the scope of any given interface of this type is generally narrow. The addition or major modification of routines controllable by the interface is beyond the abilities of its typical user.

Traditional flowchart methods are often used in both higher- and lower-level VPLs. Flow charts reduce the complexity of textual code somewhat, but can still be quite cryptic and do not efficiently use screen space. Occasionally, pictures accompany or are used in place of the text (as in Pict [5] or HI-VISUAL [7]) within a flowchart, but this does not help to give syntactic clues for programming. Nassi-Schneiderman flowcharts, used primarily for lower-level programming, are more compact than traditional flowcharts and have an implied syntax. They can be textually cryptic and difficult to read, however.

There are other VPLs which use pictures and other visual cues in order to construct the program use non-traditional flow methods. Proc-BLOX [6], a lower-level VPL, allows users to create Pascal-like code by assembling blocks representing the textual code primitives in a jigsaw puzzle fashion. The shapes of the elements preclude the possibility of assembling syntactically incorrect programs. Other packages such as Lingraphica™ [8] and ISHeE [9] remove the text altogether and rely on pictures to determine the meaning of the program. ISHeE also uses the jigsaw puzzle format to convey syntax. By making the visual representations more compact, more of the program under development can be seen on the screen at a time.

3. Details of Software Framework

A multilevel interface requires a multilevel programming framework in which to operate. Associated with our research into multilevel IPL/HMIs is the development of a multilevel reconfigurable software framework [11][14]. In this section, we introduce this software framework, and discuss its various components.

3.1. Overview

The real-time components of our software framework (illustrated in Figure 1) are sup-

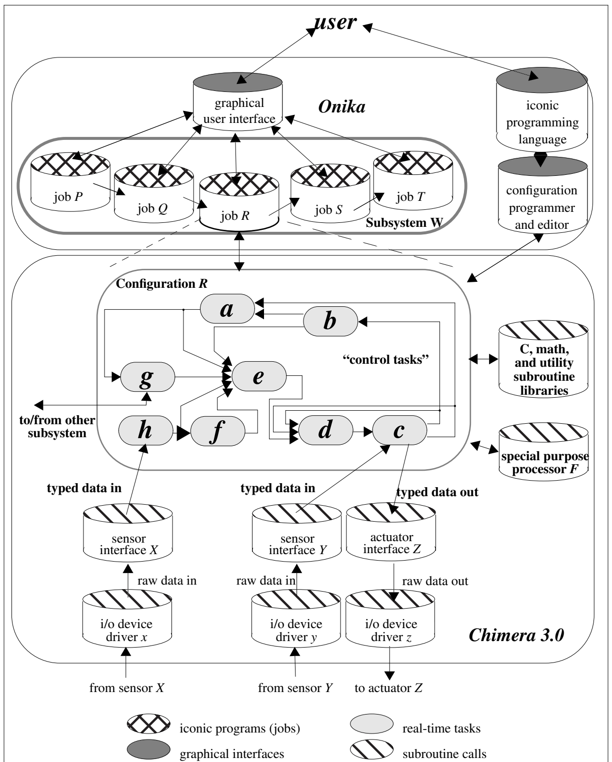

Figure 1.The reconfigurable software framework for sensor-based real-time operating systems. Routines at one level are created by combining modified reusable routines at the adjacent lower-level.

ported by the Chimera 3.0 Real-Time Operating System [12]. The user interface and programming environment for these real-time components are implemented within Onika [3][4].

We define a *control module* as a reusable software module within a reconfigurable sensor-based control system. A control module executing in the real-time environment is referred to as a *task*, and hence we often use the two terms interchangeably. Control tasks may be either *periodic* or *aperiodic*.

A *configuration* is formed by integrating control modules from a library to form a specific configuration. Device drivers and utilities (such as math subroutines) are automatically "linked in" based on the needs of each module in the configuration. A configuration implements functions such as motion control, world modeling, behavior-based feedback, multi-agent control, or integration of multiple subsystems.

A *job* is a high-level description of the function to be performed by a configuration; e.g. *move to point x*. When the post-conditions of one job and the pre-conditions of the next are satisfied, then a dynamic reconfiguration to the next job can be performed within the system. We use the term *action* interchangeably with the term *job*.

A *control subsystem* is defined as a collection of jobs which are executed one at a time, and can be programmed by a user. Multiple control subsystems can execute in parallel, and operate independently or cooperatively.

An *application* is defined as one or more subsystems operating in parallel. An application may be composed of subsystems of other applications, allowing for hierarchical decomposition of an application.

In the following sections we discuss the basic building block of our framework, the control module.

3.2. Control Modules

Each control module has zero or more *input ports*, zero or more *output ports*, and may have any number of *resource connections*. Input and output ports are used for communication between tasks in the same subsystem, while resource connections are used for communication external to the subsystem, such as with the physical environment, other subsystems, or a user interface.

Each input and output port is a state variable, and not a message port. Whenever a task executes a cycle, the most recent data corresponding to the input port variables is obtained. At the end of a cycle, the new data corresponding to the output port variables is used to update the subsystems's state information.

A *link* or *connection* (the terms are used interchangeably) is created by connecting a port of one module to a port on another module. A configuration can be legal only if every input port in the system is connected to one, and only one, output port (see section 4.3.2). An output port may connect to multiple input ports.

A task does not have to have both input and output ports. Some tasks receive input from, or send input to, the external environment or to other subsystems using the resource ports. Other tasks may generate data internally (e.g. trajectory generator) and hence have no input ports. Still other tasks may just gather data (e.g. data logger), and hence have no output ports.

The software framework described in this section allows the user to create reusable and reconfigurable real-time software. However, direct use of the operating system which supports this framework requires users to be knowledgeable about textual real-time code. For the naïve user, a novel human-machine interface is required to fully use the system. In the next section, we discuss Onika, our human-machine interface for this software framework.

4. Onika

4.1. Onika as an Interface

The purpose of Onika is to provide an appropriate interface for each level of our programming framework. Each interface shares with the other interfaces the common concept of building higher-level routines from combinations of lower-level routines. In theory, there is no limit to the number of levels of programming which can be created by such a framework. Although it would be impossible to create an interface for each potential level, it is possible to use the same interface for closely allied levels. This is particularly true at higher levels, where the routines that define an application are all goal-oriented. In Onika, we have defined the following levels of programming: the *lower level* (also called the *textual level*), the *middle level* (also called the *control level*), and the *upper level* (also called the *application level*). Upper level routines are combined into routines which are also usable in the same upper level programming environment. This means that no additional high-level interfaces are needed. Onika provides both a robot interface and programming environment for the middle and upper levels of programming. It also uses lower level programs to define middle level routines.

This next sections discuss the interfaces at each level of Onika in greater detail, including the rules for combining routines and modifiers into higher-level routines.

4.2. Lower Level Details

Device drivers and sensor interfaces are the routines of the lower level of the system's programming framework. Sensor interfaces are created by combining various device drivers, and manipulating the data which is received from and sent to those drivers. These framework elements use C code, which can be generated by using a VPL or other C-generating program (such as MATLAB), as suggested in section 4.1. Onika currently does not interact with these levels in a direct manner. Unlike higher levels, the creation of routines from these building blocks needs to be done by a technically oriented user having extensive programming knowledge and an understanding of real-time operating systems.

Device drivers and sensor interfaces are combined with other code to create control modules. It is beyond the scope of this paper to define the legality of and modifications to combinations of sensor interfaces and device drivers, and the interested reader should to refer to [11]. The use of the routines created by the sensor interfaces is discussed in the following section.

4.3. Middle Level Details

In the middle level interface, upper level routines may be created by combining certain modified routines called "tasks" into control block diagram form. Knowledge of textual coding is not required, but merely a good working knowledge of control theory.

4.3.1. Combining task routines

The basic unit of combination at the middle level is the *task*. As mentioned in section 4.1, 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 middle level user.

A parameter file is associated with each task's module. This parameter file completely describes the task. When Onika is executed, it loads in all available task parameter files on the system. 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 lexicon*. 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 4.3.2.

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 2). Onika then checks the pins of the new tasks and

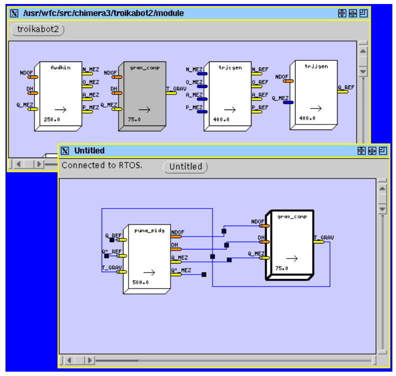

Figure 2.Tasks placed on the canvas are automatically connected to the tasks already there.

determines whether each has a similar variable name to other pins on the canvas. If so, 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 [3].

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. 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. Furthermore, a combination of tasks on the canvas can be saved at any point for later recall.

4.3.2. Task combination rules

Within a task, any state variable can be declared as any of the following: *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 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.

4.3.3. Creation of higher level routines

Before the combination of tasks can be saved as a job, there must be exactly one output instance of each state variable used in the configuration. As mentioned in section 4.3.2, this is 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 (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 an object*, whereas a job which requires no modifiers is simply an *action*. The modifier of a job is referred to as an *object*.

Once a job routine has been created, it is available for use in the upper level interface. The use of job routines in the upper level is the subject of the next section.

4.4. Upper Level Details

Similar to the middle 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 (*objects*) are displayed in the same window as the available routines. This provides an easy mechanism for modifying any given routine. Jobs (*actions*) and *objects* 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 spawned and activated. 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.

4.4.1. Combining job routines

The basic unit of combination at the upper level interface is a *job*. A job is created at the middle level by combining tasks together (see section 4.3.3. on page 6). This functionality is hidden from the upper-level user, however. A job may or may not require a modifier, depending on how it was defined at the middle level. Jobs which require modifiers are referred to as *actions requiring an object*, whereas jobs which do not require modifiers are referred to simply as *actions*. An *action requiring an object* icon must be followed by exactly one *object* icon.

An *object* icon could be created for any state variable from the global state variable table. A preference file defines the types of *objects* which Onika will recognize. *Objects* can be created at both the middle and upper levels. The user supplies both the object type and its value(s).

All icons are presented to the user in a *job dictionary*. Each icon's picture is framed in a structure which has a left and right edge of a certain shape and color. These are indicators as to which type of icon can sit next to another. Onika will not allow non-interlocking icons to be placed next to each other.

All *objects* 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.

4.4.2. 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 gap

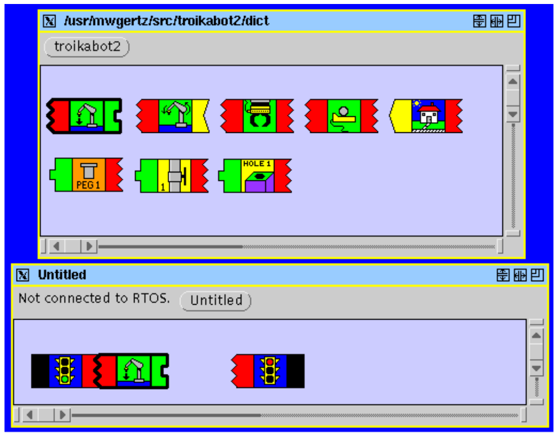

Figure 3.The icon just inserted into the application did not match with the icon following it in the application flow. A space was inserted for an *object* icon.

(Figure 3). 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.

Conditional branches, *parallel branches*, and other potential icons will introduce their own syntax needs. These constructs have not yet been introduced into Onika in any form.

Applications created by combining jobs and modifiers can have icons assigned to them and be used in other higher-level applications. Whereas "incomplete" applications (i.e. those with some object 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 an object*, provided that any gaps within the incomplete application refer to the same type of *object* consistently.

5. Summary

There has previously been little-to-no research done in the area of multilevel interfaces for reconfigurable sensor-based control systems, despite the fact the type of programming varies dramatically between the different levels of the system. Until the use of multilevel sensor-based systems becomes widespread, and the various levels of the system are equipped with programming and control interfaces appropriate to the abilities of their potential programmers, the use of sensor-based robots will continue to be narrow in focus and difficult to implement. The framework and interface presented in this paper constitute one step in the direction of achieving an completely integrated sensor-based system which will expand the usefulness of robots in laboratories, industry, and business.

Acknowledgments

The research in this paper is supported, in part, by Sandia National Laboratories, NASA, and the Dept. of Electrical and Computer Engineering and The Robotics Institute at Carnegie Mellon University. Partial funding for Matthew W. Gertz is provided by NASA Langley Research Center through a GSRP fellowship.

References

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

[2]Gertz, M. W. “The Onika User’s Manual,” (in progress) Department of Electrical and Computer Engineering, Carnegie Mellon University.

[3]Gertz, M. W., Stewart, D. B., and Khosla, P. K. “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.

[4] Gertz, M.W., Stewart, D. B., and Khosla, P. K. “An Iconic Language for Sensor-Based Robots,” in Proceedings of SOAR Conference, August 4-6, 1992, Houston, Texas.

[5] Glinert, E. P. and Tanimoto, S. L. “Pict: An Interactive Graphical Programming Environment,” *Computer*, November 1984, pp. 7-25.

[6] Glinert, E. P. “Out of Flatland: Towards 3-D Visual Programming,” Proceedings 2nd Fall Joint Computer Conference, 1987, pp. 292-299.

[7] Ichikawa, T. and Hirakawa, H. “Visual Programming – Toward Realization of User-Friendly Programming Environments,” Proceedings 2nd Fall Joint Computer Conference, 1987, pp. 129-137.

[8] 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.

[9] 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.

[10] Myers, B. A. “Taxonomies of Visual Programming and Program Visualization,” *Journal of Visual Languages and Computing*, 1990 (1), pp. 97-123.

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

[12]Stewart, D. B. and Khosla, P. K. *Chimera 3.0 Real-Time Programming Environment*, Program Documentation, Dept. of Elec. and Comp. Engineering and The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213 (e-mail *chimera@cmu.edu* for a copy).

[13]Stewart,D. B., Schmitz, D. E., and Khosla, P. K. “The Chimera II real-time operating system for advanced sensor-based robotic applications,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 6, pp. 1282-1295, November/December 1992.

[14]Stewart, D. B., Volpe, R. A., and Khosla, P. K. “Design of Dynamically Reconfigurable Real-Time Software using Port-Based Objects,” Technical Report CMU-RI-TR-93-11, Dept. of Elec. and Comp. Engineering and The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213.