1994

Visual Programming and Program Visualization Techniques for Port-Based Objects

Matthew W. Gertz
Carnegie Mellon University

Pradeep Khosla
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/isr
Visual Programming and Program Visualization Techniques for Port-Based Objects

Matthew W. Gertz and Pradeep K. Khosla
Dept. of Electrical and Computer Engineering
The Robotics Institute at Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

In this paper, we discuss in detail the motivation behind the creation of the Onika Iconic Programming Language and Human-Machine Interface, developed at Carnegie Mellon University and used to program and control various robotic systems in both autonomous and teleoperative modes. Additionally, we discuss the syntax and grammar of the iconic languages presented therein, and show how they can be used to create complete applications to be executed on any manipulator.

I. INTRODUCTION

The development of software for reconfigurable sensor-based real-time control systems is a complicated and tedious process, requiring highly specialized skills in real-time systems programming. Recently, however, a software framework has been introduced to reduce the development time and costs associated with real-time programming [12]. This framework introduced the concept of port-based objects (objects that have input and output ports for real-time inter-object communication), which allow real-time software to be reused and reconfigured. While this framework greatly reduces the difficulties involved with real-time application programming, its use as a stand-alone paradigm limits its effective scope to only those with in-depth knowledge of textual coding and real-time systems, who may not in fact be the end-users of any applications developed. The scope of its use, however, can be greatly expanded by the introduction of a novel visual programming environment which incorporates both visual programming (VP) and program visualization (PV) techniques. The Onika Iconic Programming Language and Human-Machine Interface, a hybrid VP/PV programming environment, was developed at Carnegie Mellon University to decrease the time and expense required to create portable software applications for robotic manipulators [3]. Onika presents both a control-oriented engineer’s interface and goal-oriented general interface, in which task or job elements represented by icons are dragged to an assembly window and arranged into syntactically-correct programs which, barring gross physical dissimilarities, can be executed on any manipulator connected to the system, both locally and remotely [5].

In this paper, we discuss in more detail some of the motivation behind creating such a system, and describe in detail the grammar and syntax of the iconic language resident in both the engineering and general Onika interface.

II. MOTIVATION

We have been driven in this research into visual programming environments by the need to allow both naïve and knowledgeable users to create reusable and reconfigurable real-time code using port-based objects. In particular, the resulting interface had to meet the following criteria:

• The interface must be able to display and use conventionally-coded procedures to create higher-level code.
• The programming interface must be devised in such a way that it need not be changed when new port-based objects are introduced into the object library, or when old port-based objects are modified.
• The syntax for combining the procedures must be clear and expandable.

These motivations are discussed in greater depth in the following sections.

A. A New Method of Visual Programming

Graphical interfaces fall within one of two categories: visual programming (VP) or program visualization (PV), both of which are subclasses of visual languages [8]. 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. However, in a reconfigurable system, the user must be able to
manipulate pre-existing dynamic arbitrary procedures 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 [8], 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 a reconfigurable system. 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. These ideas are implemented within our software framework and within Onika, and are discussed in more detail in Section IV.

B. External Subsystem Interfacing

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 would require changing only the code of the port-based object itself, leaving the interface unchanged (see Figure 1). Such a scheme would significantly reduce the time required to support new types of displays and sensory inputs. We have implemented such a external subsystem communication mechanism within our reconfigurable software framework, which we discuss further in Section V.

C. Dynamic 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. As mentioned in Section II.B, such rewrites are expensive and can lead to poorly-written code.

To solve this problem, a mechanism needs to be introduced by which new target and procedure types can be added to the VP interface without needing to rewrite the code. This would also allow for the modification of target information as well. Such a mechanism has been developed and implemented within Onika, and is discussed in Section VI.

III. TERMINOLOGY

Our software framework for reconfigurable R&A systems has been discussed extensively in [3] and [12]. In this section, we briefly review some of the terminology associated with our framework.

A control module is an instance of a class of port-based objects. Details of port-based objects are given in [12]. 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.

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. 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.

A task library is a directory which contains task files and the control module object code. There can be multiple task libraries opened for concurrent use within Onika.

A configuration is formed by connecting tasks from a library to form a specific routine; for instance, a robot task, gravity compensation task, differentiator task, and trajectory task might be connected to form a joint motion configuration. In Onika, tasks are automatically connected graphically by comparison of the names of their inputs and outputs as they are placed into configurations. In a complete configuration, no task has a hanging input. Configurations can be saved for future use.

A job is a high-level port-based object, which refers to a specific configuration. 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. Certain jobs may not require any such target input, but may be self-contained. In Onika, jobs are rendered as mnemonic pictures, appended and prepend-ed to which are visual cues specifying the type of job.

A target is also a high-level port-based object, which supplies the data required by a job. Their graphical aspects 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 instance) locations in space, but can be as simple as a numeric index to use in an internal case statement, or a filename to which data can be written.

1. Also referred to as an object in our previous papers; we use the term target in this paper to avoid confusion with the term port-based object.
Targets can be modified at any time, except during execution. Their visual presentation within Onika is similar to that of the jobs.

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.

A control subsystem is a collection of actions which are executed one at a time, and can be assembled by a user.

An application is one or more subsystems executing in parallel. These subsystems can include control subsystems based on our software framework for reconfigurable systems, as well as subsystems based on other software frameworks, such as vision subsystems, path planners, neural networks, and expert systems, to name a few. Applications may contain other applications, allowing for the creation of routines of even higher levels. Applications may have conditional paths, and can be saved for later recall.

IV. TASK PROGRAM VISUALIZATION

Tasks are the lowest-level type of routine with which a general user might interact. In this section, we discuss the different types of tasks in our framework, and the qualities which a general task possesses.

A. Task Aspects

The code for control tasks in our software framework for reconfigurable systems is small and portable. It is written in a conventional manner, external to the real-time operating system and the visual programming environment [12]. It is the responsibility of the visual programming environment to display the aspects of the task in such a way as to make its function completely understood by the user. While it may be impossible to display all task aspects at once, the user is always able to interact with a task icon to view any of its aspects (for instance, using mouse clicks to indicate that the certain parameters of the task should be displayed). The Onika interface for interacting with tasks is shown in Figure 2.

Certain task aspects are fundamental; that is, they refer to basic information which the user cannot change from within the visual programming environment. These aspects include the visual indication of a task (a graphical box, circle, or some other consistent shape which represents a generic task), the number and types of I/O ports associated with the task (though not the names of those ports), the name of the task, the code which it executes, etc. Other aspects are mutable and can be changed; these include the frequency or period of the task, the state of the task, and the names assigned to the ports (the latter are especially necessary when sharing code between two sites which use different naming conventions).

In our framework, it is not useful to parse compiled module code to retrieve the aspects of a task for several reasons: the compiled code does not provide defaults for such aspects as CPU and task frequency, the relationship from task to module code is not one-to-one, and the encoded names of the I/O ports may not match the naming conventions assigned to our own system. Therefore, associated with each task is a parameters file, which allows the user to specify the passive aspects of the task, as well as defaults for certain active task aspects. Using the information in task parameters files, it is trivial to construct a block icon to represent a given task. Information which is too lengthy or too abstract to
display visually can be displayed with mouse clicks. For instance, in Onika, the user can display (and, to a certain extent, modify) a task’s parameter file and display its module’s C-code (if available) by clicking on the task’s icon in a various manners.

B. Types of Tasks

In our framework, tasks are combined together in some logical manner to form action primitives (such as “Move To Point X” or “Open Gripper”). When the job which these tasks perform is complete and correct, there needs to be some mechanism by which the controlling system is informed, so that the next job in line can be performed. Only one task in any job configuration should be able to signal the controller that the job is finished; otherwise, the system may be in an unknown state at the termination of the job, or the controller may receive two or more termination signals. Since a termination signal might also contain information as to how the next job should be performed, the subsequent actions to a job would be unpredictable if more than one task could terminate the job. We have devised the notion of a trigger task to pass such signals to the control system.

All tasks need to have the ability to signal the controlling system in case of error, so the dichotomy between trigger tasks and more general tasks is largely one of convention. We assume that a job will reach some state where it is can terminate with no errors reported; for instance, when a goal has been reached, or when a certain amount of time has expired. Thus, the logical candidate to be the trigger task in a “Move To Point X” job would be the module which generates the trajectory of the motion, since it would know when the point had been reached. The trigger task in an “Open Gripper” job would be the one module which actually passed the “open” signal to the gripper hardware. When these tasks note that their goal has been achieved, they then send the controller (Onika, for instance) a signal package indicating that the job has been completed. The signal package can also include information as to the degree of success of the job, which, when analyzed by the controller, can be used to determine how to proceed. Other non-trigger tasks would only pass signals to the controller if they encountered errors. The error signals can be analyzed by the controller, and appropriate actions can be determined.

There are certain other tasks which, even if they fail, do not affect the outcome of the job. For instance, consider a task whose sole job is to send the current configuration of a robot to a CAD visualization program. This task only outputs data, and does not introduce any data which would affect the other tasks which actually move the robot. It would be inappropriate to halt the motion of a robot performing a critical job simply because the CAD visualization program crashed, causing the data-sending task to go into error. We therefore conceive the notion of a passive task. When a passive task goes into error or needs to turn off, it signals the controller, which can deactivate that specific task without affecting the other tasks in any manner, and without losing system stability. This has the added benefit that a motion job containing the aforementioned data-passing task can be used to drive a robot even if the user decides not to display the motion on the CAD program. Note that a passive task can never be a trigger task, since the “job finished” signal from the trigger task will cause the controller to deactivate all other tasks in a stable manner.

The distinction between tasks into general, trigger, or passive is largely one of programming. General tasks only send error signals, trigger tasks send either error signals or “job finished” signals, and passive tasks only send deactivation-request signals. The task programmer should determine into which category a new task falls, and program the task appropriately.

In the next section, we discuss another ability available to all tasks: the ability to connect with external subsystems.

V. EXTERNAL SUBSYSTEMS

In the previous section, the passive data-passing task mentioned as an example required external subsystem communication. As mentioned in Section II.B, a mechanism for external subsystem communication is important as it frees the system designer from rewriting the programming interface every time a new system must be supported. Furthermore, the programming interface does not need to stop monitoring the job execution to deal with the nuances of the external subsystem. In our framework, we assume that any communication between a job and a specific external 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.

To gain the most benefit from the use of external subsystems, there should be no limitations on where the external subsystem can run, how fast it must operate, or on what platform it must run. The only requirement should be a mechanism for communication between it and the outside world. Communication between processes which may be running on different systems is generally done by sockets; either over a local communication line, or over some other communication path such as the Internet.

Within our framework, 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 socket (over 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, through error, or by the controller at job completion).
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 the programming interface 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.

In this section and the previous section, we have dealt exclusively with the creation of jobs (such as motion primitives) from reconfigurable tasks. In the next section, we discuss how a graphical grammar can be used to combine these jobs to create any program which might be created by more conventional methods.

VI. VISUAL GRAMMAR

The grammar of conventional textual programming languages for robots tends to fall into three categories: motive, structural, and numeric. Motive language constructs include such examples as move arm <a> to <x> with duration=<t> or drive joint <j> of arm <a> to <q> with duration=<t>. Structural language constructs examples include if...then...else, while...do, and cobegin...coend. The engines which process numeric language constructs are generally fundamental and are used for the purposes of calculation; e.g., \( i = i + 1 \) or \( P = V^*I \). In this section, we discuss how these three constructs are implemented within Onika.

A. Motive Constructs

Motive constructs are those which directly interact with a robot. They include all motion commands, gripper commands, and pause commands. Typically, these require some parameters indicating which arm should be affected, how long it should be thus affected, and what its end state should be. The commands are hard-coded into the language, which limits the programmer to certain types of motions (generally Cartesian or joint motion which is absolute, relative, or along some polynomial path).

Within Onika, the actual motions are jobs which have been built from tasks (we sometimes refer to these jobs as actions). Thus, the type of motion is not limited to a few choices. If a new type of motion is required (for instance, “move to x with torque limits of v”), the appropriate job can be designed from tasks and made available to the programmer.

The parameters to the various jobs are passed via targets. Each type of job requires a certain type of target, the data structure of which is defined in some preference file. For instance, a joint motion job might require the desired joint position and the duration of the trajectory to that point to be in its associated target. By changing the data associated with the target icon (perhaps by clicking on the icon in a certain way to bring up a target editor), we can change the location to where the robot will be moved without actually rewriting any motion code. The pictorial icons representing the jobs and targets are visually displayed in such a way as to make it immediately apparent which targets can be connected with any particular job (in Onika, we color- and shape-code the edges of the icons). Of course, not all jobs require targets; “open gripper” would be an example of such a job. Figure 3 shows how the jobs and targets can be assembled into an Onika application.
B. Structural Constructs

Certain structural constructs are rather easy to implement in a graphical fashion. Parallel flow, normally indicated by something like \texttt{cobegin...coend}, can be indicated simply by placing the two streams of code beside each other on the screen. \texttt{Begin...end} blocks are similarly easy. Conditional code and looping code is somewhat more difficult to visualize, due to its non-linear nature.

In the case of conditionals, each trigger task, when signaling the end of a job, also sends in its signal package some indication as to how the application flow should proceed. For instance, if a vision job fails to find a certain object in the environment that it would normally expect to find, its trigger task might signal the controller to pass control to the “else” branch of a following \texttt{if...then...else} construct. More basically, one might create a job to analyze current state conditions and return certain values in its signal package based on what it finds. This information can then be used to determine which branch to take. Such a job could even have a target, giving the job’s trigger task some values against which it can compare.

The return signal is also analyzed to determine whether \texttt{while...do} loops should be exited or not. Two jobs can be assigned as the beginning and ending of the loop, with an indication as to whether or not the loop should be \texttt{top test} or \texttt{bottom test} (this determines the exact time at which the return packet is analyzed for looping data). The job whose return signal is thus tested may perform some motive function, or may simply be a counter or other job designed solely for analysis.

A structural construct missing from most textual robot languages is the \textit{synchronization marker}. In textual languages, if two arms are to be synchronized, the user must use a series of \texttt{cobegins} and \texttt{coends}, applying pauses of certain durations to the arms at appropriate times, and hope that his addition was correct and that the times of the two arms matched up properly. With synchronization markers, such guesswork is eliminated. The user instead selects the icons of two jobs which are in two different parallel flows of an application, and applies a marker to each. During execution, when one marker is reached, the flow in which it was encountered pauses until the other flow has reached its corresponding marker. Both marked jobs then begin executing at the same time. A synchronized application created in Onika is shown in Figure 4.

C. Numeric Constructs

In our software framework for reconfigurable systems, most numeric calculations are confined to the tasks comprising the jobs. It is the responsibility of a task to calculate force from torque, for instance. In the instances where a counter variable is necessary (for instance, to determine whether a while loop should be exited or not), a job can be created which takes, as a target, a number indicating the amount of looping that is to be done. The job can keep track of how many times it has been executed since the last while loop was entered, and the job’s trigger task can signal when the loop should be exited.

The most common numeric construct in our framework is the \textit{target}, discussed in Section VI.A.

VII. FUTURE WORK

The visual programming and program visualization techniques discussed in this report and implemented in Onika are currently in use on several different systems within our laboratory and elsewhere in the United States and Canada. This has allowed us to reduce training time and programming time on our manipulators from weeks to days, or even hours. We have also been able to port module code to other sites without needing to recompile or rewrite it in any way.
Additionally, since Onika connects to the Chimera real-time system via the internet, we have successfully controlled robots remotely using Onika from distances up to 2,600 kilometers away [5]. Despite these achievements, there are certain areas within our visual programming environment which require further research and development. These include:

- **User Testing:** Onika is currently undergoing user testing using standard prototyping techniques as given in Meister [6]. User feedback from standardized testing is essential for designing an interface usable by non-specialists.

- **Loop implementation:** Currently, while...do loops are implied by “tagging” two icons as the “start” and “end” of the loop, with the assumption that the job represented by the test icon (“start” or “end”) will return some indication of whether the loop should be continued or not. This implies that the test job must be “special” in some way, requiring a high-level user to create the lower-level job and trigger task. The test conditions are further hidden from the high-level user. A better visual way to implement test conditions at the higher level should be developed for Onika.

- **Conditional implementation:** Case blocks are currently saved in separate applications, which are “iconified” and included in higher-level applications. The return code of the preceding icon’s job is analyzed to determine the case to follow. As with while...do loops, this also implies that the analyzing job is “special.” Furthermore, since the case block is reduced to the size of an icon, it is needlessly difficult to read the code. Finally, the conditions under which one case is taken as opposed to another are buried in the trigger task’s code. All of these problems need to be addressed.

Additionally, the Onika programming environment should be expanded to support more than one real-time operating system (currently, only the Chimera 3 Real-Time Operating System is supported).

**ACKNOWLEDGEMENTS**

The research reported in this paper is supported, in part, by the National Aeronautics and Space Administration (NASA), Sandia National Laboratories, the Department of Electrical and Computer Engineering, and The Robotics Institute at Carnegie Mellon University. Partial support for Matthew W. Gertz is provided by the NASA Langley Research Center through a Graduate Student Researchers Program fellowship.

**REFERENCES**


