The First World Congress on Intelligent Manufacturing Processes and Systems, February - 1995.

A Visually Intensive Lifecycle Framework for Robotic Applications

Nicholas Tarnoff, Albert J. Wavering and Ronald Lumia*
Building 220/ Mail Stop B127
National Institute of Standards and Technology
Intelligent Systems Division
Gaithersburg, Maryland 20899 USA

Abstract

While the direct manipulation of geometric models is well established for robot trajectory planning, robot program logic is still developed textually and provides no visual run-time feedback of its own. The original goal for this project was to enhance traditional Off-Line Programming technology with the visual programming of discrete robot tasks. In contrast to relatively application specific iconic visual programming languages (VPL), we explored a scalable VPL that makes little use of application specific symbols. Visual logic and geometrical information complement each other to provide a simple robot programming interface for the end-user.

The use of visual programming was eventually expanded beyond the original goals of the project. Scalable visual logic was applied both throughout the control system and seamlessly across the system lifecycle. Object oriented principles were also applied to help structure and generalize system components. Results of this project include an object-oriented visually intensive framework for developing discrete-time control systems and an example application of this framework to dextrous inspection. New robotic applications are built by an engineer using a scalable visual language to extend robot control base classes and form a fluid control architecture. The end-user also uses the same scalable visual language to specify, simulate and execute task level robot plans.

Keywords: Visual programming language, robot control, graphical simulation, software architecture, object orientation.

1. INTRODUCTION

Off-Line Programming (OLP) is a widely used method of programming and simulating robot motion. Solid geometry models provide a visually interactive method of specifying robotic motions without the hazards or costs of using real equipment. An important component of OLP is geometric information. Geometric information is migrated from CAD design systems to the OLP environment where robot layout, reach, collision, task logic and cycle-times are simulated. Once a robotic task is programmed and tested in simulation, the task is executed on-line. The geometric information then becomes part of the on-line control system world model [1]. OLP is therefore part of the CAD geometry's lifecycle process where CAD seamlessly evolves from design through operation. This

---

Note: U.S. Government work is not subject to photocopy restrictions within the United States. References to commercial products within this paper do not imply a recommendation nor that the products are the best for the purpose.
* R. Lumia: Mechanical Engineering Department, Room 202, University of New Mexico, Albuquerque, NM, USA

lifecycle leveraging of geometric information reduces the complexity and cost of building and evolving control systems.

In addition to manipulating geometric information, OLP requires control logic. Control logic is used to program the coordinated behaviors of actuators and sensors. Unlike visual geometric information however, the control logic of traditional OLP systems is textual and not explicitly animated. Logic information must be inferred by looking at the motions of solid models. This geometric emphasis to robot programming inadvertently de-emphasizes the logic (conditions, parallel operations) portion of the robot task. The goal of this project, therefore, is to make task logic explicit and graphic in order to increase attention to its importance in specifying new robot tasks. Toward this goal, we are developing and integrating methods for interacting with program logic that are similar to the process of interacting with CAD graphics.

We focused on a visual programming language (VPL) that is highly scalable because it relies very little on application specific symbolic labels. Iconic differences are largely reserved for representing fundamental language components. Application specific functions are differentiated primarily using text. The engineer uses the VPL to interactively extend the software framework for new applications. In addition, the end-user can use the same VPL foundation to interactively develop robot task logic using a set of application specific instructions (task vocabulary). Robotic tasks are layered by combining lower-level high-resolution instructions to form subsequently higher-level lower-resolution instructions. The application engineer typically builds layers of robotic instructions up to a level below which the end-user will operate. The resulting layered combinations of instructions form a task hierarchy [2].

Of several applications, dextrous inspection is the last and most mature application developed within the proposed visual software framework. The end-user is presented with a set of robotic VPL instructions which can be combined to form an inspection task. This inspection task in the form of visual logic is coupled with traditional geometric specifications made directly on graphical solid geometry. New inspection tasks require changes primarily to geometric information but very few changes to the visual logic. The visual logic is altered primarily for new robotic applications of the framework or for logic intensive robotic tasks.

In section 2, the paper delineates the primary technical areas of study, their scope and related work. In section 3 the paper describes the inspection application and how the end-user tool is used to interactively specify, simulate and execute an inspection task using both traditional geometry and visual logic. In section 4, the object-oriented software, control and implementation architectures are presented. In section 5, the paper presents the role of visual logic in supporting system wide control behaviors.

2. SCOPE

2.1 Visual Programming Languages (VPL) for the Man-Machine interface

Traditional man-machine interfaces (MMI) or graphical user interfaces (GUI) consist primarily of customizable widgets such as symbolic icons, menus, dials, and charts. A visual programming languages is also an important part of the MMI but differs from traditional interface components. A VPL involves the use of directly manipulable and generic visual structures such as box and line (graph) languages. As such, visual programming languages are a class of visual formalisms [3]. As general programming languages, visual formalisms take advantage of human vision's high bandwidth, pattern recognition and random access abilities. This project explores the use of visual programming over traditional graphical user interface components.

A scalable VPL was explored in order to minimize application specific symbolism. Iconic differences are reserved for the language's fundamental constructs such as boolean, instantiation or branching operations. In contrast, many VPLs make greater use of symbolic icons to simplify the recognition of operations [4]. Symbolic languages such as these are not currently addressed but could be supported using scalable languages by attaching symbolic icons to otherwise textually differentiated icons. Such a hybrid iconic facility would enable the system to support an even greater variety of VPL vocabularies.

Initially the project focused on improving the interactive specification and verification of robot tasks by the end-user. The early benefits of visual programming (as described in the conclusion) led to its expanded role. Visual programming was eventually applied to many levels of control as described in the implementation architecture.

2.2 Object Orientation

Visual programming functionality for this project was supported by Prograph 2.5. Prograph is a commercial object-oriented development environment. Object Oriented class hierarchies were applied to all components of the control system as compared to the GISC system [5] where objects are used primarily for equipment level control interfaces to actuators and sensors. The class definitions provide a reusable information modelling framework for subsequent applications. In addition, the visual representation of class hierarchies helped in understanding the overall system and in identifying generalizable components across branches of class trees. Despite their importance, concurrency and control architecture information were not explicitly represented graphically for this project. Concurrent behavior is not explicitly managed in our system as do some object oriented tools through the use of state charts [6] [7]. Instead our system uses a recursive scheme (section 4.1) to interleave control node processes.

Object-oriented system typically include memory management facilities such as garbage collection. Garbage collection supports the run-time creation and destruction of objects. Traditional software languages are also used to accomplishing memory management tasks particularly where performance tuning is needed to support high frequency sensory interactive servoing [4]. Object-oriented systems, however, provide explicit services and language constructs that encourage on-line variability in the structure of algorithms. To this effect, our system uses a recursive scheme to traverse a hierachy of otherwise independent control node objects.

3. APPLICATION

3.1 Dextrous Inspection Workcell

Dextrous inspection was chosen in order to support the efforts of the Auto Body Consortium project in improving the dimensional quality of car body assemblies. Dextrous inspection provides flexible and in-line inspection. In-line inspection is more effective than off-line inspection because in-line inspection measures 100% of parts and provides immediate feedback about up-line problems or down-line solutions. Measurement accuracy, however, is lower than with traditional coordinate measuring machines.

The visual programming tool is designed to simplify the task of specifying robotic inspection tasks for the end-user. Geometric information is inherently intuitive as a direct representation of objects. Visual logic was added to the interface to further simplify the task of specifying task logic in a syntax neutral way. The inspection workcell shown in Figure 1a consists of a Robotics Research Corp. (RRC) arm and LVDT touch probe. Figure 1b illustrates the traditional geometric representation of the workcell used to interactively specify robot trajectories.

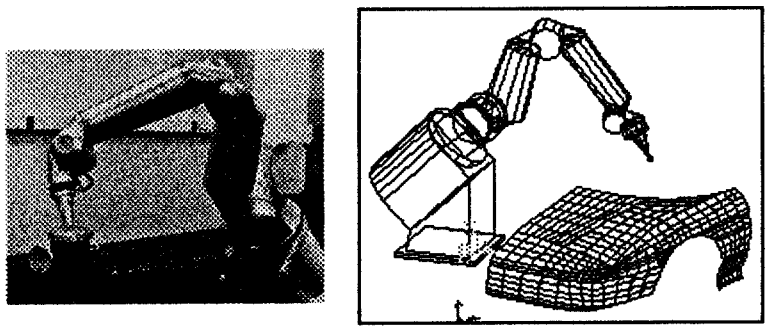


Figure 1(a) - Actual workcell, (b) - robot and car panel models

Unlike traditional OLP systems, however, the robot task logic (conditionals, parallel operations, loops, etc.) is an equally noticeable graphical and interactive part of the user interface. As described in section 4.3, the geometric information and visual logic are displayed on different monitors. Figure 2 provides and example of the visual logic display for the end-user. This example represents an inspection plan that is animated during execution and provides the end-user with a clear understanding of the logical system state as a supplement to traditional geometrical state information. This visual logic can be interactively paused, modified and resumed by the end-user.

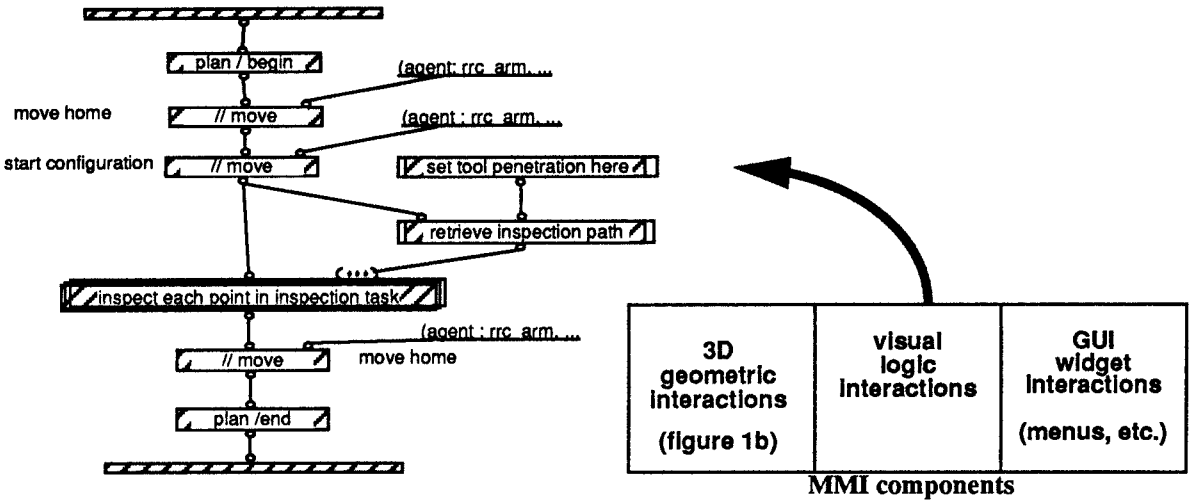


Figure 2 - Visual logic of inspection plan

3.2 The Man-Machine Interface

The end-user uses visual logic as a simple (no syntax, clarity of logic flow) highly interactive language for designing, testing and executing robot tasks. Execution may be paused and resumed at any time during interpreted execution. The logic can be modified during a pause and execution is resumed just before the modified code. The mouse is used to add new instruction icons or change flow connections. This visual programming framework is based on the Prograph 2.5 language. This language provides all the constructs of a full featured language such as C but in the form of data flow graphs. Iterations between design and execution are modeless in that the granularity of each iterative cycle is completely adaptable by the developer. Transitions between each mode (editing, execution) take no significant amount of time compared to traditional compilation cycles. This language foundation was used to build objects specific to robot control. In addition, application specific instructions are encapsulated in the form of new textually differentiated icons. The developer implements an application specific set of instructions which can be assembled by the end-user. The set of instructions available to the end-user in this way is called the task vocabulary for a specific application.

The visual combination of robot task instructions by the end-user is not subject to a static validation prior to execution. The validity of connected instructions is evaluated at run-time. This approach is similar to that of textual languages with no static type checking. Systems such as these are allowed greater run-time variability in the connections of VPL constructs but will require greater care in operational testing. To minimize operational testing requirements, the developer implements some dynamic type checking. For example, algorithms are added to critical robot instructions to check incoming data flows at run-time. This embedded validation of data flow provides increased system safety while preserving end-user flexibility.

3.3 Inspection Task User Scenario

The GUI dialogue consists of menus and prompts that guide the user in specifying inspection tasks. The inspection task specification consists of both geometric and logical information. First the user selects inspection trajectories by picking points directly on the 3D CAD model of the part as shown in Figure 1b. These points are immediately checked for static reachability. Static reachability is a fast and preliminary way to check for reach by checking wether the robot can reach each inspection point based on a common starting configuration. Further checking of reachability is performed at run-time to establish the reach of inspection points in their task specific sequence. Once the inspection points are selected and checked, they are combined with the inspection plan logic shown in Figure 2 and tested in simulation. Unlike traditional OLP system, the plan logic is graphically animated during execution to provide the end-user with a clear and rapid understanding of both the entire inspection task and the current execution state. The visual logic is responsible for retrieving inspection point geometry at the plan step labeled "retrieve inspection path". The inspection points are then fed to the inspection subroutine "inspect each point in inspection task". Potential collisions and dynamic reachability are checked at run-time within the geometric simulation. The visual logic on the other hand runs interpreted and may be paused, modified and resumed at any time. This level of iteration with the task logic provides greater control over the process of specifying, testing and executing a robot task.

Once the inspection trajectory and task logic are deemed satisfactory, the inspection task (geometry and logic) is automatically translated and downloaded to the lower level robot control system. The RRC robot will execute the inspection task and display its activity in real-time through the 3D graphical display. Resulting inspection measurements are imported into the simulated world and are viewed in 3D near their respective nominal inspection points.

4. SOFTWARE FRAMEWORK

4.1 Reference Control Architecture

Control functionality for the system described in this paper is organized according to the RCS hierarchical reference architecture [2]. Figure 3 illustrates a general instance of this architecture.

Figure 3 shows an example full-featured design whereas the control hierarchy for this project's implementation (presented later) is much simpler. This reference architecture has proven very useful in organizing complex manufacturing, autonomous vehicle and other robotic systems. Control nodes consist of Sensory Processing (SP), World Modelling (WM) and Task Decomposition (TD) processes. TD in turn contains Job Assignment (JA), Planning (PL) and Execution (EX) modules. The architecture provides powerful and generic guidelines for organizing a hierarchy of control behaviors.

A basic understanding of this architecture is helpful for understanding latter sections of this paper. This control structure is reflected in the low level control of the RRC arm and touch probe, higher level visual control logic, class hierarchies, communications and control behaviors [8].

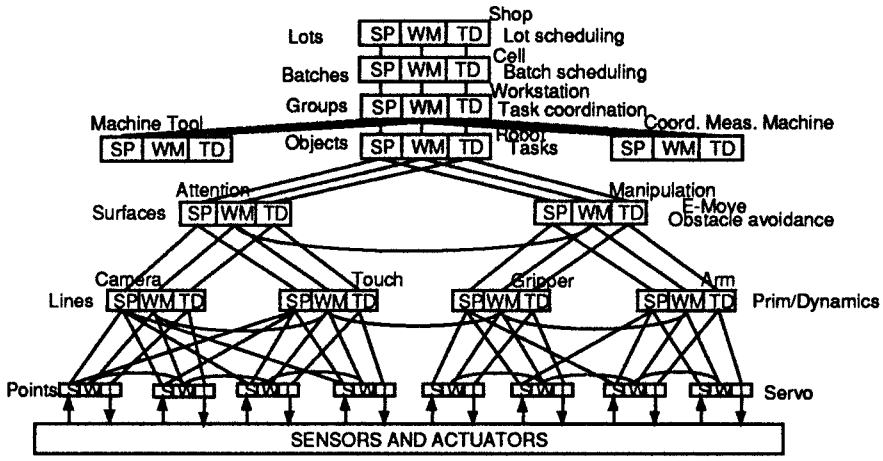


Figure 3 - Example Full-Featured RCS Control Architecture

4.2 Application Control Architecture

The inspection application required primarily discrete elementary move (e-move) and task level control (3rd and 4th levels). These mid-levels of control are interfaced with existing RCS primitive (prim) and servo control modules [9,10]. Task level control, for example, involves planning around statements like: "inspect part A" or "assemble parts A and B". Task level plans then decompose these task commands into commands that are issued to the lower emove control level. Commands to the emove control level typically refer to features on a part. The plan in Figure 2 is one such plan for inspecting a series of points on the surface of a part. This transformation from one control level to the next involves retrieving world model data such as inspection points, part locations and part features. This world model data is used to specify motion command parameters or to make decisions about what action to take next.

Sensory processing is responsible for processing, abstracting and storing real-time feedback for use in planning and execution at each control level. The e-move control level of the inspection application, for example, accepts the measured inspection data and stores it together with the original nominal inspection points. The nominal and measured inspection points are now available within the world model for further processing and evaluation. The existing primitive and servo levels of control as described in [10] also use sensory processing.

The application specific control architecture in Figure 4 follows reference architecture guidelines. The design of a control hierarchy typically begins with primarily "top-down" analysis of the task. The potential task decomposition is iterated against a "bottom-up" process and resource analysis. For this application, the primary task is to inspect a part. The resulting hierarchy for this application is a rather straightforward coupling of the robot and probe resources with the inspection task. In addition, the level control node is added to decompose the inspection task among multiple features on one part. The resulting command vocabulary is shown between control levels. The Prim and Servo modules are labeled with "real or simulated" because control nodes at various stages of development can be connected to test the system incrementally. For example, the real Prim and Servo levels can drive a dynamic simulation of the robot activity. In another configuration, Prim an Servo are simulated from within CimStation (commercial OLP tool) to drive a graphical simulation of the plant. The real Prim and Servo can also drive the plant simulation. In contrast, the Emove and Task levels of control evolve in a more seamless fashion through their lifecycle (design -> simulation -> operation).

The operational control hierarchy is fluid. Messages between control nodes are targeted at run time. No persistent connection exists between any two control nodes. For this implementation, however, the small number of lower level resources means that the control hierarchy is relatively static.

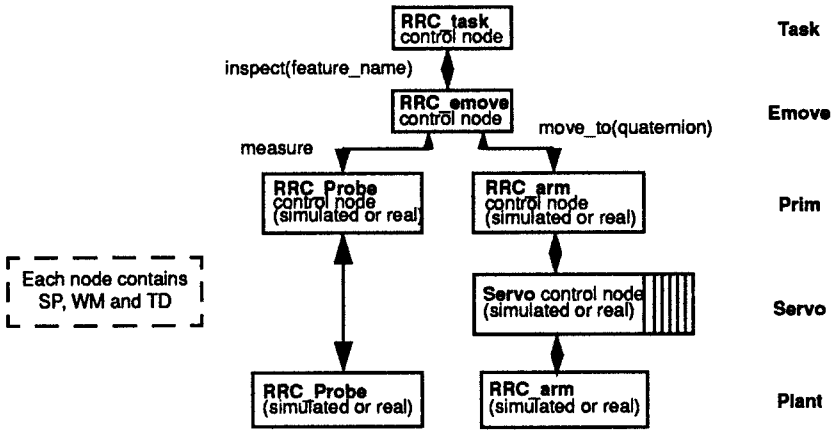


Figure 4 - Application specific control architecture

4.3 Implementation Architecture

Figure 5 summarizes the major software tools, functional components and communication channels of the implementation. The dynamic and servo control modules are covered in [9,10].

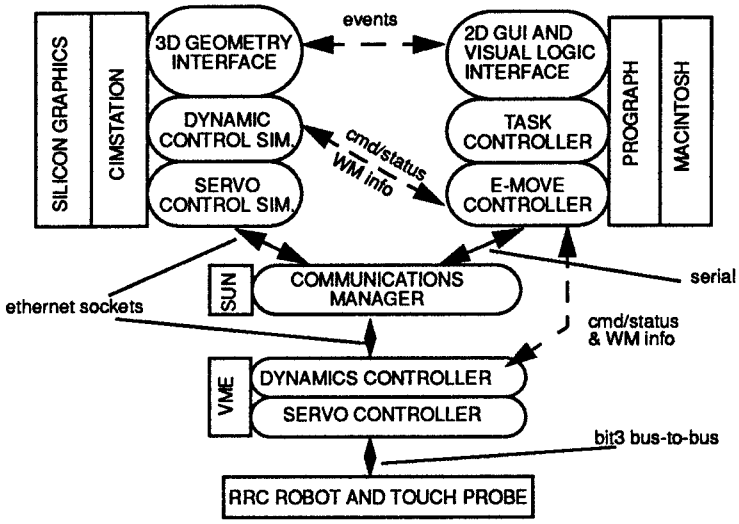


Figure 5 - Major tools, functional components and communications.

The two major components of the "user interface" include three dimensional (3D) geometric information and two dimensional (2D) information each managed primarily by CimStation and Prograph respectively. These two components are managed by different applications and, for this implementation, run on different computers. The two interface components are coordinated through the exchange of events. The events transfer control between the two displays to support a constrained interaction script for inspection. The distributed user interface required that we develop additional functionality within CimStation to support communication with a remote user interface and to script 3D graphic user interactions.

4.4 System Classes

Object-oriented facilities are an integral part of the Prograph language and were applied to all control and user interface components running on the Macintosh. The Emove, Task and user interface components shown in the upper right hand corner of Figure 5 were implemented using the major classes shown in Figure 6. These class structures are the result of an iterative development process

whereby higher level specifications are increasingly stable with each subsequent application and lower level branches are used to capture more immediate and localized changes to the system. Some familiarity with the object-oriented concepts of polymorphism, encapsulation and inheritance is helpful for understanding the class structure described below.

Command frames encapsulate the command vocabulary between control nodes. While some specializations are specific to the robot (RRC task, RRC emove) other command classes remain more generic (prim move, prim grip). Generic classes are used wherever possible. Characteristics that are specific to an application or to specific equipment may require a specialized subclass. This subclass may in turn become a relatively generic class with subclasses of its own. The process of structuring new classes is still very much an art. However, adding new classes is much clearer in a system such as this one where several stable levels of classes have already been developed.

The parameter classes are most often used to encapsulate parameter data that is part of a command frame. The *cartesian_parameter* contains all of the information passed into the primitive control node for operating the RRC robot. The *cartesian_parameter* attributes are: *cartes_space_algorithm, cartes_goal_position, pose_offset, goal_pose, world_ref_frame, endeff_ref_frame, redundancy_resolution, compliance_goal, goal_force, force_offset, destination_object,* and *traversal_time*. The *inspection_path* parameter is used to encapsulate inspection path information within the RRC primitive control node world model. The *universal_parameter* is used to instantiate parameters where type information is part of the attribute list rather than part of the class specification. This is convenient for simple or short-lived data type objects that do not require their own class.

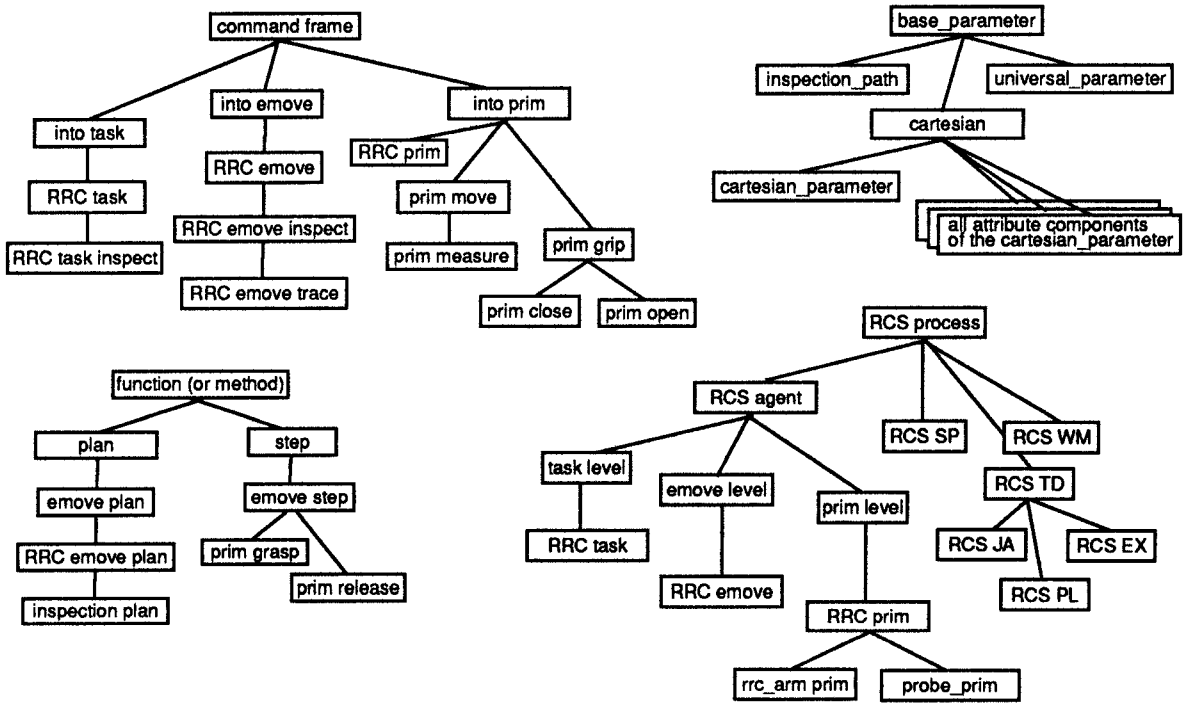


Figure 6 - Major class hierarchies for the RRC workcell

The plan and step classes are specifications of available plans and plan steps. The actual plans and steps are stored in an other class hierarchy (not shown) that functions as a persistent directory structure for storing methods. This approach is necessary to define classes that are primarily logical (rather than data) oriented. Conventional object-oriented systems do not support the storage of methods as objects themselves. Ideally an object-oriented system should function like a database where object "attributes" consist of anything from traditional data type objects to logical-type objects.

The RCS process hierarchy specifies virtual process classes (also called active objects or agents). These objects are used to construct the control hierarchy shown in Figure 4. Currently, the TD behaviors are completely generic. Future applications may require control level, control node or application specific specializations in which case the TD class hierarchy will be extended.

5.0 SYSTEM BEHAVIOR

Traditional object-oriented classes do not explicitly express dynamic system behaviors. Two major behaviors are therefore described below.

5.1 Recursive Servo Loop Scheduler

The Servo Loop Scheduler is a data flow process whose responsibility is analogous to that of an operating system scheduler. The Scheduler subdivides and executes otherwise logically concurrent RCS processes including SP, WM, JA, PL and EX (defined on page 5) One grouping of this set is called a control node. In addition, the scheduler orders the execution of these processes to reflect the behavior of a servo control loop as shown in Figure 2. This scheduler is then recursively nested in order to traverse the hierarchical branches of an RCS control tree such as the one in Figure 4.

Figure 7 represents the Prograph data-flow program for cycling through SP, WM, JA, PL and EX. This program is recursively nested and the arrows which enter and exit the dashed contour represent the entrance and exit channels of the recursion.

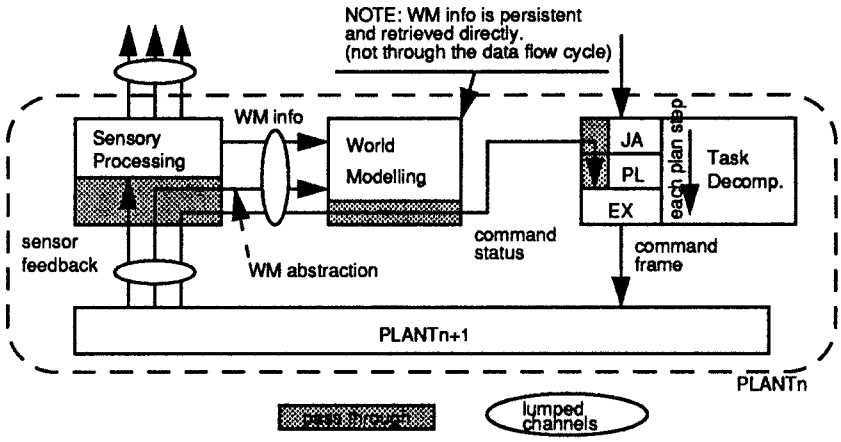


Figure 7- Recursively Nested Servo Loop

The Scheduler subdivides and interleaves the execution of these logically concurrent RCS control node processes according to a certain granularity. For this system the granularity is event driven rather than time based. Command frame messages are the controlling events. In other words, each of the control node processes will be cycled once for each command frame issued by the same node. The recursion, therefore, traverses the run-time COMMAND tree. The functional nature of this interleaving process means that interleaved segments do not take the same amount of time to complete. Rather, the relative execution time spent executing different control nodes will vary slightly depending on the time required to complete one inter-level command (or event).

This execution schedule may provide a relatively complete solution for executing concurrent RCS modules on one physical CPU. Theoretically, a fast CPU could apply this recursive schedule to a set of concurrent RCS modules and ensure that the real-time requirements of the lowest control levels are met and that subsequently higher levels of control are executed relatively fewer times as suggested by the reference architecture

5.2 Task Decomposition Planning

Task decomposition involves three potentially pipelined functions: Job Assignment (JA), Planning (PL) and Execution (EX). The job assignment allocates tasks among multiple subordinate agents. The Planner is responsible for producing (planning) coordinated plans of action for each subordinate agent. Finally the executor executes the plan produced by the planning module and integrates feedback

Planning is largely performed off-line by the end-user. On-line planning functions for this system involve changes in data but not in the plan structure. The plan structure captures a mix of information that spans the scope of all three functions (JA, PL, EX). For example, a particular plan step may require a specific subordinate executor in which case the job assignment is hardcoded. As another example, a visual plan (tree diagram) with joining flows can specify a planned synchronization point. The plan in Figure 2 is an example plan where some JA, PL and EX functions are an integrated part of the plan. In addition, base classes for JA, PL and EX encapsulate generic behaviors that form the system design.

In our system, if a plan step does not contain the information necessary for its execution (such as target agent or world model data) then JA, PL and EX apply their generic methods of completing the step DATA. Plan LOGIC, on the other hand, cannot be modified by JA, PL or EX. Plans are written in Prograph and no facility currently exists for computationally modifying the logic of Prograph programs. Plan data is completed one plan step at a time. Plan steps are identified and passed through JA, PL and EX as shown in Figure 4. The identification of plan steps from a Prograph plan is the only reflection available. Each plan step leads to one command frame (but may lead to many). The execution of JA, PL, EX and other concurrent RCS modules are therefore segmented and interleaved recursively around each plan step such as those shown in Figure 2.

6. CONCLUSION

Intelligent manufacturing systems are becoming increasingly difficult to manage effectively as the number and diversity of integrated components continue to grow. Standardization and unification provide means of simplifying the integration of complex systems by industry. This project has increased our understanding of how a scalable visual language can be used to integrate different system across their lifecycles. Most importantly, this understanding will be used to promote the use of commercial visual languages as a unifying technology and to support the unification of general and domain specific languages to further industry's progress in managing complex intelligent systems.

Visual logic is an intuitive method of providing the non-programmer a method of capturing and observing the logic of robot plans at various levels of control. The visual logic for an inspection task, for example, is an explicit part of the interactive graphical interface. The entire task is made clear and the currently executing step is highlighted. This logical information is not clear from looking at only geometric animations and static program text. The logic information is also relatively free of syntax. Interpreted and animated execution alleviates long compilation cycles and promotes a modeless development approach. Visual logic is also a very good functional compliment to more common visual languages for creating concurrent agents and describing state machine behavior. Visual logic is also broadly useful for its scaling and lifecycle properties as illustrated in Figures 8 and 9. Geometry is a more familiar concept that exhibits similar properties.

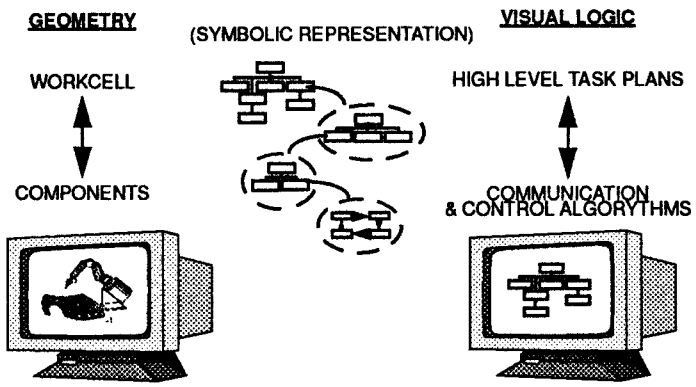


Figure 8 - Unified System Detail (spacial unification)

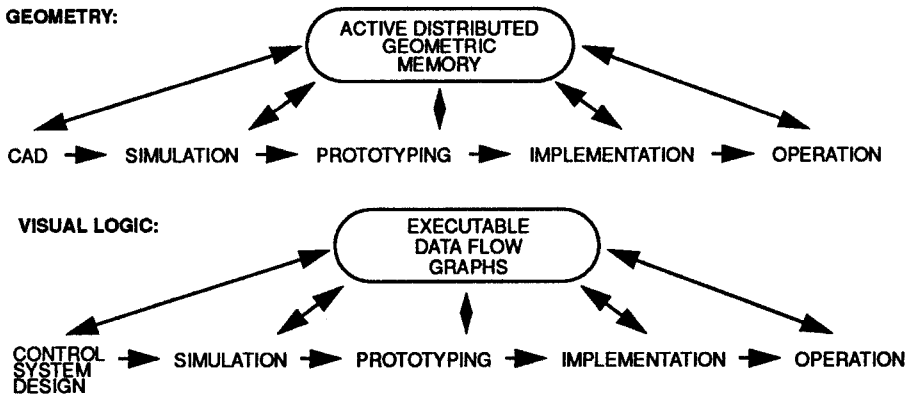


Figure 9- Unified system lifecycle (temporal unification)

7. FUTURE DIRECTIONS

We will continue to explore and possibly unify scalable and evolutionary visual constructs for other domains and other systems dimensions such as concurrency and behavior. We intend to support general purpose and domain specific visual programming languages such as those listed in the IEC-1131-3 Programmable Controller Programming Languages standard (Ladder Diagram [LD], Function Block Diagrams [FBD] and Sequential Function Charts [SFC]). In addition, visual language constructs might be unified to cover an even greater range of intelligent manufacturing operations for both discrete and continuous processes [11].

The framework described in this paper emphasizes the use of geometry, hierarchy (control architecture and classes), and visual programming of functional algorithms. These "meta-quantifications of complexity" [12] provide powerful methods for understanding and building complex systems. These methods also have in common their strong dependence on visual and scalable representations for their understanding, manipulation and improvement. The following future directions integrate control and software architectural principles and will in turn benefit from the visual representations discussed in this paper.

7.1 Reflective Planning:

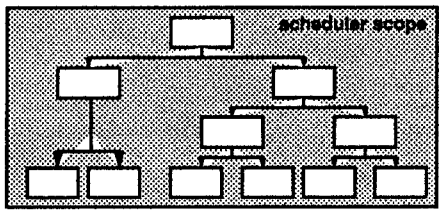
Computational planners (non-human) rarely manipulate non-linear logic as part of their planning duties. Complex plan logic is therefore often pre-determined manually. Completely automated planning, however, will require the manipulation of logic. Manipulating logic is counter to traditional computing methods where data and function are separate and only data is changed at run time.

In order to modify or generate plan logic on-line, access is needed in the same way that data is traditionally manipulated at run-time. One solution, therefore, is to represent plans (including complex logic) as manipulable data structures. The approach taken by expert systems, for example, is to store rules and then chain these rules according to a pre-determined decision scheme. The expert systems, however, are rarely written in their own language. A more unified solution would involve using the same language for planning as for representing plans themselves. Broad computational reflection would provide this ability to modify Prograph plans, for example, using the same language [13].

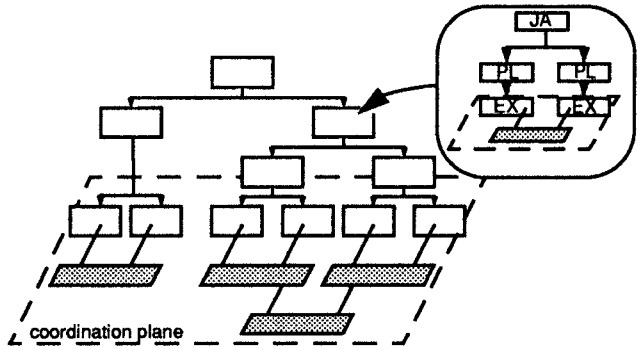
Computational reflection also leads to the possibility of developing many levels of planning reflection. Each resulting plan at a particular level of reflection would in turn act as a planner for yet another planning level. The last planning level would produce a non-reflective plan for execution. Multiple levels of reflective planning are similar to the abstract layering of hierarchical control levels. Reflection might therefore provide an appropriate language between behavior generation modules of lower resolution. Low level high-resolution control modules can transition to more efficient message based communication.

7.2 Coordination hierarchy

System Architectures are often characterized and limited by relatively planar representations. The following structure, however, illustrates how complex visual geometries can help express and build advanced system architectures. First consider how an open operating system (OS) can be customized to exhibit a specific behavior such as scheduling. This customization may take a very specific form such as to coordinate the execution of two specific control nodes. A hierarchy of such OS-like schedulers form a coordination hierarchy as pictured in Figure 10(b). At the lowest level, this orthogonal behavior plane can coordinate groups of devices and sensors [14]. Other orthogonal (or less than orthogonal) hierarchies can be used to synchronize the execution of two or more plans (inset). In general various degrees of orthogonal behavioral planes can be used to effectively manage several focused but cooperating behavioral hierarchies.



10(a) - one OS scheduler



10(b) - multiple schedulers = coordination hierarchy

REFERENCES

Note: The National Bureau of Standards (NBS) became the National Institute of Standards and Technology (NIST) in August, 1988. The Robot Systems Division became the Intelligent Systems Division in June, 1994.

[1] N. Tarnoff and R. Lumia, "The Role of Off-Line Robot Programming in Hierarchical Control", Second International Symposium on Robotics and Manufacturing Research, Education and Applications, ISRAM, Albuquerque, NM, November 1988.

[2] J. S. Albus, "An Introduction to Intelligent and Autonomous Control", Chapter 2: A Reference Model Architecture for Intelligent Systems Design, Kluwer Academic Publishers, 1994.

[3] A. N. Bonnie and C. L. Zarmer, "Beyond [mental] Models and Metaphors: Visual Formalisms in User Interface Design", Journal of Visual Languages and Computing, 1993, Vol. 4, pp. 5-33.

[4] Matthew W. Gertz, David B. Stewart and Pradeep K. Khosla, "A Software Architecture-Based Human-Machine Interface for Reconfigurable Sensor-Based Control Systems", 8th IEEE International Symposium on Intelligent Control, Chicago, Ill, August 1993.

[5] J. M. Giesmeyer, M. J. McDonald, R. W. Harrigan, P. L. Butler, B. Rigdon, "Generic Intelligent System Control (GISC)", Version 1.0, Oct. 1992, Sandia Nat. Lab. Report: 92-2159.

[6] "Object-Oriented Analysis and Recursive Development and The Intelligent OOA Tool", Ipsys Software and Kennedy Carter product literature, London, UK, 1993.

[7] B. Selic, G. Gullekson, J. McGee and I. Engelberg, "ROOM: An Object-Oriented Methodology for Developing Real-Time Systems", CASE'92 Fifth International Workshop on Computer-Aided Software Engineering, Montreal, Quebec, Canada, July 1992.

[8] N. Tarnoff, A. S. Jacoff and R. Lumia, "Graphical Simulation for Sensor Based Robot Programming", Journal of Intelligent and Robotic Systems, 1992, Vol. 5, pp. 49-62.

[9] R. Lumia, "Using NASREM for Real-Time Sensory Interactive Robot Control", Robotica, 1994, Vol. 12, pp. 127-135.

[10] A. Wavering, "Manipulator Primitive Level Task Decomposition", NIST Technical Note 1256, NIST, Gaithersburg, MD, December 1989.

[11] A. D. Baker and D. K. Carter, "I-I-CON: A Visual Programming Language for Integrated Industrial Control", 1994 International Programmable Controller Conference, Detroit, Michigan, April, 1994.

[12] A.B. Çambel, "Applied Chaos Theory: a paradigm for complexity", Academic Press, 1993.

[13] S. Matsuoka, T. Watanabe and A. Yonezawa, "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming", European Conference on Object-Oriented Programming (ECCOP '91), Geneva, Switzerland, July 1991.

[14] L. Acar and U. Ozguner, "An Introduction to Intelligent and Autonomous Control", Chapter 4: Design of Structure-Based Hierarchies for Distributed Intelligent Control, Kluwer Academic Publishers, 1994.