2001

Development of simulation and off-line programming software modules for 5-axis waterjet cutting gantry robot.

Aleksandar Z. Boskovic
University of Windsor

DEVELOPMENT OF SIMULATION AND OFF-LINE PROGRAMMING SOFTWARE MODULES FOR 5-AXIS WATERJET CUTTING GANTRY ROBOT

by

Aleksandar Z. Boskovic

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through Industrial and Manufacturing Systems Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada
2000

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-96403-5
Our file Notre référence
ISBN: 0-612-96403-5

ABSTRACT

The topic of this dissertation is development of the software modules for simulation and off-line programming of 5-axes waterjet cutting gantry robot. In order to verify and validate the quality of the written software modules, a 5-axes gantry robot manufactured by Flow Robotics Company and equipped with an Allen-Bradley 9 series controller has been used for testing. The mentioned simulation and off-line programming software modules are not stand alone applications, therefore they are dependent on the software platform they are written for. In this case, Workspace 5® software package serves the purpose of being the platform for the written add-ins.

In order to get a proper simulation, the robot controller programming language – G code, has been translated to a common format to be used internally by the simulation package. The robot native language has been translated first into C and then to Workspace Simulation Language code, in order to call the motion planning routines through the Component Object Model (COM) interface. Flex and PRECCX together form a compiler-compiler tool used for translation. A supporting C file has been created to add the flexibility to the way PRECCX compiler works.

A Realistic Robot Simulation (RRS) set of services has been implemented in the default motion planner of the simulation software to execute the robot's functions and behaviors in the same way a robot controller does. Due to a fact that Realistic Robot Simulation is primarily designed for 6-axes vertically articulated robots, several Realistic Robot Simulation services must have been modified and a preprocessing graphical interface made in order to reflect the kinematics differences from 5-axes gantry robots.

To make a powerful and effective simulation, a workpiece has been designed that is complex enough to require 5-axes machining. Also, a CAD model of the gantry robot has been designed and the appropriate kinematics template associated with it.

The reverse process – off-line programming, has also been developed. Having known the robot's trajectory and motion parameters, the off-line programming software module has been developed to automatically emit G-code part programs into designated output files. This module communicates with other modules of the simulation package like default motion planner, simulation engine, and default kinematics in order to retrieve the information needed for accurate formatting of the output code.

This thesis had its practical verification in Flow International Corporation. The results of preliminary testing showed that the recorded value difference between the position and orientation values (both Joint and Cartesian) of the teachpoints created during simulation and the Joint and Cartesian values read from the robot controller was satisfactory. Also, simulation and real robot cycle time accuracy has been determined comparing the corresponding cycle time values recorded in Workspace and in the robot controller. The cycle time accuracy has been assessed as satisfactory and acceptable, because the error value was below the acceptable upper limit set by Realistic Robot Simulation standard.

DEDICATION

DEDICATED TO MY FAMILY

LIST OF ABBREVIATIONS

AGV – Automated Guided Vehicle

ASCII – American Standard Code for Information Interchange

AST – Abstract Syntax Tree

CAD – Computer Aided Design

CAM – Computer Aided Manufacturing

CAR – Computer Aided Robotics

CNC – Computer Numerical Control

COM – Component Object Model

FLOW – Flow International Corporation

GP – Geometric Point

IGES – Initial Graphics Exchange Specification

OLP – Off Line Programming

OOS – Object Oriented Simulation

PC – Personal Computer

PRECCX – Prettier Compiler-Compiler Extended

RRS – Realistic Robot Simulation

SAT – Save As Text graphics exchange format

VBA – Visual Basic for Applications

VR – Virtual Reality

VRC – Virtual Robot Controller

WS5 – Workspace5®

WSL – Workspace Simulation Language

YACC – Yet Another Compiler-Compiler

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

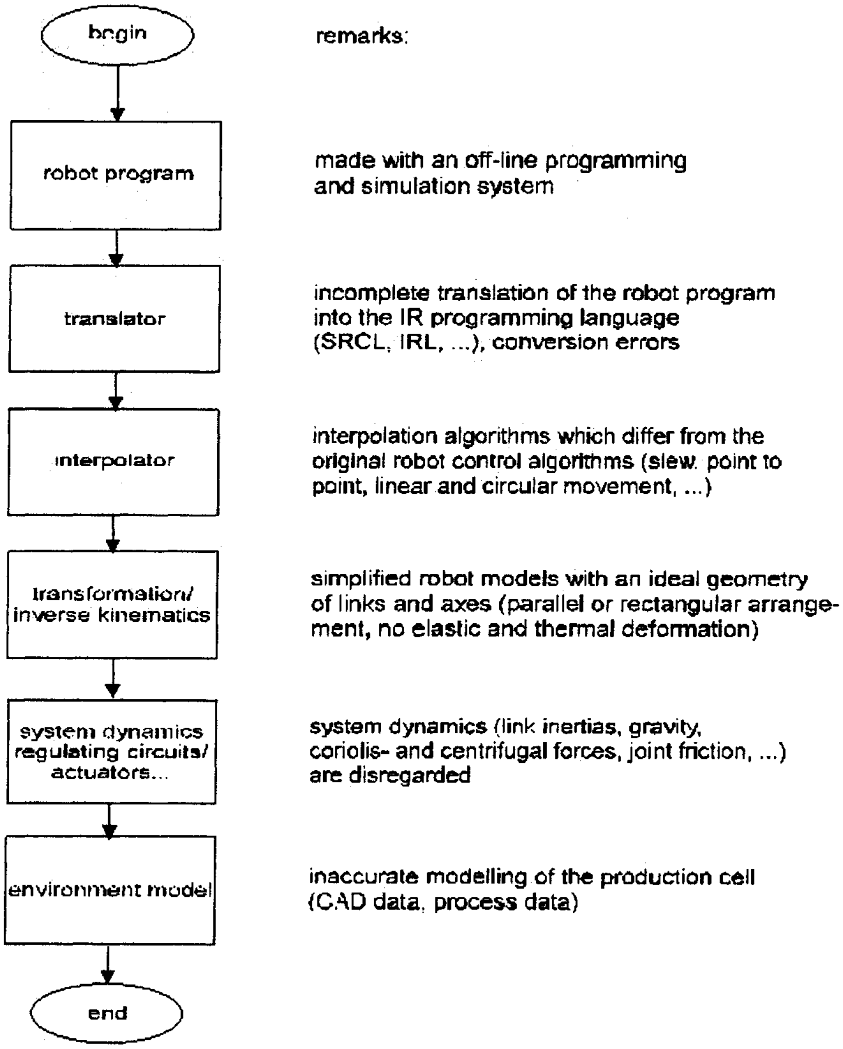
1.1. THE THESIS OVERVIEW

The installation and operation of a robotic manufacturing system frequently proves to be a much more expensive venture than initially planned or imagined. Robotic systems are expensive to purchase, install, and operate. Unrealistic expectations based on technical equipment specifications may lead to errors in the robotic cell and application design. Such errors are very expensive and difficult to rectify once the robotic equipment is selected, purchased, and installed.

The operation of robots is also frequently less efficient and effective than expected. The main advantage of robots is their flexibility (the ability to easily change the programmed tasks, regardless of how different and how complex they are) which allows their implementation for manufacturing a variety of products in small and medium batch sizes. Flexibility is essential in modern manufacturing to respond to short product life cycles, varying demand, small production lots, and model changes. However, productivity is much easier to achieve in a dedicated mass production system, and few flexible systems are also highly productive. One of the key reasons for low productivity in robotic systems is the fact that robot programming requires the allocation of a considerable amount of robot production time, both for program development and for testing.

CAD (Computer Aided Design)-based graphic emulators and simulators of robotic systems can potentially help to avoid some of the roadblocks on the way to successful robot system installation and operation. Cell design, robot selection, verification of robot reach and of correct placement of the cell elements, off-line programming, and simulation of the robot task can all be done in a virtual CAD and simulation environment. Simulation models that accurately represent the proposed robot and cell geometry and the robot kinematics and dynamics performance are valuable tools for evaluating design alternatives, verifying feasibility, designing workcell layout, verifying robot programs, and evaluating cell performance.

Compared with the advantages during the optimization of robot cell layout or during collision and reachability testing, there are difficulties in transferring user programs that have been generated and simulated off-line to the real production. Different sources of errors during simulation and off-line programming have been depicted on the block diagram below (Figure 1.1.), followed by more detailed explanation of each error source.


begin
robot program
translator
interpolator
transformation/ inverse kinematics
system dynamics regulating circuits/ actuators...
environment model
end

remarks:

made with an off-line programming and simulation system

incomplete translation of the robot program into the IR programming language (SRCL, IRL, ...), conversion errors

interpolation algorithms which differ from the original robot control algorithms (slew, point to point, linear and circular movement, ...)

simplified robot models with an ideal geometry of links and axes (parallel or rectangular arrangement, no elastic and thermal deformation)

system dynamics (link inertias, gravity, coriolis- and centrifugal forces, joint friction, ...) are disregarded

inaccurate modelling of the production cell (CAD data, process data)

Figure 1.1. Sources of errors during simulation and off-line programming (the flow chart portion of this figure), followed by more detailed explanation of each error source (the description section of this figure).

Interpretation of the robot programming language

The first group of errors often occurs in the language processing of the system, the so-called interpreter or translator. One reason for this is that the extent and semantics of the language differ between off-line programming system and real robot controller. A typical example for this is the description of fly-by records (distance or velocity approximation records). Conversion or even simple syntax errors may seriously affect the execution of a user program. A fundamental condition for the transfer of simulated user programs is a post-processor, which is free of syntax errors.

Motion interpolation

Interpolation is used, for instance, in the motion modes: point-to-point, linear and circular, as well as, in the transition between different motion modes (fly-by). Interpolation can be described mathematically by numerous parameters (distance, velocity, and time parameters) and again exhibits its own variations in interpolating orientation. Even large industrially established off-line programming systems may not be able to cope with this large number of different algorithms with their fundamental path planning functions. Thus deviations cannot be avoided between simulation and real application in terms of:

- Cycle time,
- Path accuracy,
- Path velocity, or
- Behavior near singularities.

Inverse kinematics

Calculation of the inverse kinematics is performed by ideal, simplified kinematics models. Deviations in the lengths of the robot axes and assembly errors are disregarded. The effector load, thermal influences or gear elasticity affect the static compliance and decrease the absolute accuracy of the robot. Even small changes in the axis angles may seriously affect the angular configuration and may cause collisions of the robot with peripheral components.

System dynamics

A large number of parameters affect the controller systems behavior, such as mass, inertia, coulombic and viscous friction as well as elasticity. The system dynamics is generally disregarded in the simulation owing to the lack of dynamic data or because of inflexible dynamic interfaces.

Environment model

Simulation errors, in this part of the error chain, are caused by inaccurate modeling of the manufacturing environment. The following has to be taken into consideration:

- Modeling of the tools and workpieces.
- Arrangement of positioners, workpieces, tools, industrial robots, and other cell components in relation to the world coordinate system.
- Process modeling (such as: coating, grinding, cutting, etc.).

Efficient off-line programming and simulation system does not aim to eliminate all of the above-mentioned error sources. It would be very time-consuming and expensive to reduce the errors in the kinematics chain by a reduction of manufacturing tolerances. Practice-oriented off-line programming and simulation system requires at least:

1. An efficient post-processor, which is free of syntax errors.
2. A realistic motion interpolation for applications in which path accuracy or the determination of cycle time plays an important role.
3. Suitable, user-friendly calibration algorithms to avoid costly teach-in corrections, (Nof, 1999).

Taking into account the mentioned three requirements, the master thesis topic was defined accordingly. The first two requirements have been fulfilled by developing off-line programming and simulation software modules (based on the new concepts introduced in Realistic Robot Simulation specification, in order to significantly increase the accuracy of simulation software packages) in form of add-ins to an existing simulation and off-line programming software package. Workspace5® software package has been chosen to be the development platform, due to its modular architecture and its calibration software module, therefore satisfying the third requirement.

Waterjet cutting manufacturing process will be used for on-setting validation and verification of the developed software modules. The important reason for choosing such a manufacturing process is that waterjet equipment is ideally suited to robotic applications because it is lightweight, highly flexible, multidirectional and readily adaptable to pedestal or gantry systems.

In more detail, technical advantages of waterjet cutting technology include:

- *Lightweight*. A typical waterjet cutting assembly, including a nozzle, nozzle support and associated high-pressure water delivery components, weights 4 kilograms and reaction load is less than 5 kilograms. This is important, because the factor in determining if particular robot can be adopted for waterjet cutting is the robot's load capacity. The abrasivejet cutting system has a typical reactive load of less than 15 kilograms, which many industrial robots can effectively handle.
- *Adaptability.* A wide array of swivels, fittings, tubing, and coils allows high-pressure water to be delivered to the "wrist" of a robot without inhibiting or hampering its useful range of movement. Equipment has been successfully integrated with 5-, 6-, and even 7-axis robots.
- *Multidirectional Cutting.* A high-pressure jet of water can be moved in any direction across the material being cut. It does not have to cut in a straight line. It can cut extremely tight curves and inside corners and "starting holes" are generally not required.

Therefore, the thesis (as an arguable statement) elaborated in this document is *the use of Realistic Robot Simulation interface in conjunction with PRECCX parser generator in robotics simulation and off-line programming software packages can substantially improve the positional and cycle time accuracy of a 5-axes waterjet cutting gantry robot.*

The dissertation is organized as follows:

1. ***CAD modeling of the workpiece and the 5-axis gantry robot.*** Kinematics definitions of the CAD models.

2. ***Development of the simulation module.*** This is the central and the most complex part of the dissertation. The first sub-task is the development of G-code (CNC controller programming language) language translator. Compiler-compiler software utilities Flex and PRECCX (Prettier Compiler-Compiler Extended) have been used for lexical analysis and parsing of the G-code language. Parser action statements have been written to emit Workspace Simulation Language (Visual Basic for Applications based simulation language, which contains calls to the functions of the motion planner via Component Object Model RRS interface) code. Also, all the Realistic Robot Simulation services called from the parser's action statements have been designed and implemented in the motion planner module of the simulation platform software. The preprocessing graphical interface has been developed in order to capture the internally stored robot controller information that is not provided in the part programs, but presents a mandatory input for accurate simulation.
3. ***Development of G-code off-line programming module.*** Knowing all the path parameters defined during simulation and taking into account the grammar of the G-code programming language, the C++ source code has been written to recast all the path data into sequence of Computer Numerical Control machine instructions.

The next flow chart (Figure 1.2.) shows the interactions among the modules written by the author and the modules of the software package built in the development platform. Different background colors represent different levels of participation (color mappings are contained within the figure caption).

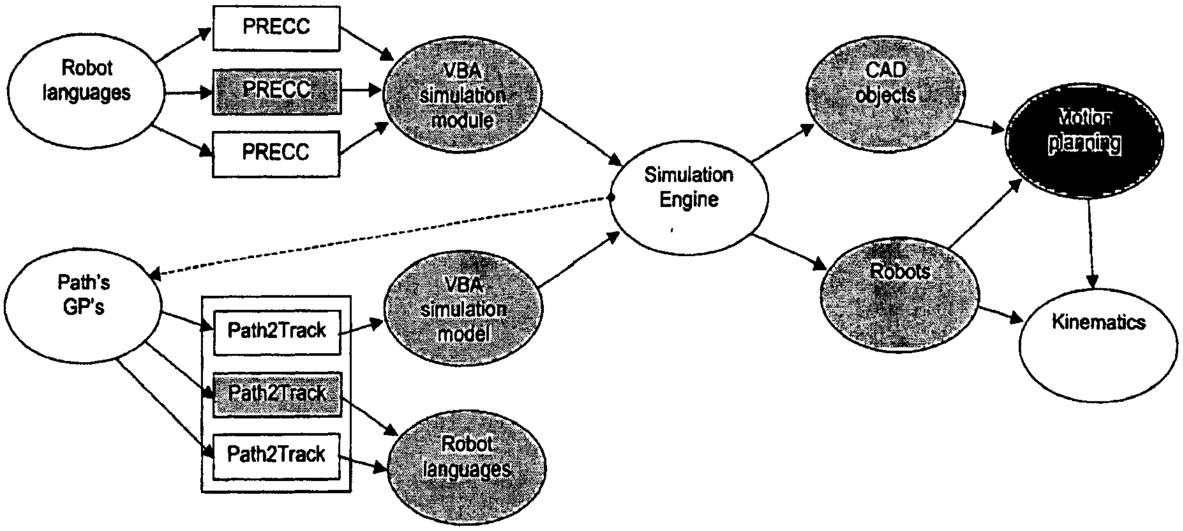


Figure 1.2. Simulation and off-line programming modular concept of Workspace5, where different colors represent different levels of author's participation (white – no participation, light gray – input modules, gray – complete design and implementation, dark – partial design and implementation)

1.2. THE THESIS RELEVANCE

Relevance of the issue of this thesis will be presented through the next three major points:

- ***Highly accurate simulation due to implementation of Realistic Robot Simulation services***. One major problem in achieving exact simulation of the robotic cell is the availability of a model of the robot controller. The algorithms defining the robot's motion behavior are not publicly available. To overcome this problem a consortium of automotive companies, controller manufacturers, and simulation systems manufacturers initiated the Realistic Robot Simulation project. It aimed at integrating original controller software (black box) into simulation and off-line programming systems via the specification of an adequate interface. The project goal was to improve the simulation accuracy of industrial robot simulation systems in order to achieve more realistic simulation of robot controllers. The goal was achieved by the definition of a common Realistic Robot Simulation interface for the integration of controller simulating modules into simulation systems. Using original controller software parts, controller manufacturers provide simulation modules for their latest controller types. Strictly following the standardization rules in the Realistic Robot Simulation specification default set of the RRS services has been implemented in default motion planner of the simulation software. In other words, when the task programs written in robot languages developed by manufacturers who did not participate in the Realistic Robot Simulation consortium (including CNC manufacturers) need to be simulated, the default set of Realistic Robot Simulation services (implemented in the simulation software source code) will be called to provide the accurate motion of the simulated robot. The Realistic Robot Simulation interface has been tested on software and hardware platforms used for robotic simulation in the automotive industry and has demonstrated impressive results of accurate simulation of motion behavior, robot kinematics, and condition handling, (Willnow et al., 1996). It has been proven that the deviation between simulated and real joint values is less than 0.001 radians, (Willnow et al., 1996). The ideal case, of course, would be if there was not any difference between the simulated and real joint values. However, taking into account that non-Realistic Robot Simulation simulation software packages have on average approximately 10 times lower angular accuracy (Nof, 1999), Realistic Robot Simulation showed considerable improvement. Concerning task cycle times, a difference of less than 3% could be reached, (Willnow et al., 1996). Again, in comparison with the non-Realistic Robot Simulation systems where the cycle time difference is in range of 5% to 10% (Nof, 1999), Realistic Robot Simulation systems are obviously better, but in this field additional improvement can still be made (Realistic Robot Simulation II interface promises 99.5% cycle time accuracy and 99.9% joint accuracy, which still remains to be proven).

- ***The first industrial application of currently the most efficient compiler-compiler utility - PRECCX.*** The use of programs intended for compiler generation facilitates the work on translator construction, so a programmer does not have to be concerned about techniques and computer limitations during translator implementation. According to the conducted literature and Internet Web search, this will be the first industrial application of PRECCX software utility. Making a compiler by using some of the contemporary programming languages (mainly C and C++) is attainable, but it takes too much time. Alternatively, using PRECCX for the same purpose will be faster and less prone to errors and also, it will have some advantages over the most frequently used compiler-compiler utilities – YACC (Yet Another Compiler-Compiler) and Bison (section 3.1.).

- ***CNC controller simulation based on the customization of the robotic simulator.*** Computer Numerical Control (CNC) machines can be considered gantry robots as far as their kinematics is concerned, but programming language that they use (often called G-code language) has its own characteristics that differ from the robot programming languages. A typical example is the definition of circular interpolation. Therefore, using Realistic Robot Simulation interface to simulate CNC programming language has required considerable amount of work, in depth knowledge of CNC controller software and Realistic Robot Simulation specification. Mapping CNC to default Realistic Robot Simulation instructions did not necessarily have one to one matching due to limited compatibility between the interface and the language.

1.3. APPLICATION IN AUTOMOTIVE INDUSTRY

This thesis had its practical verification in Flow International Corporation. The results of preliminary testing are given in the Chapter VI.

Most of Flow's gantry robots with waterjet equipment have Allen-Bradley 9-series controllers. The company needed a simulation and off-line programming software product, which would have been delivered together with their CNC and robot equipment to the customers. That software package must be capable of simulating the pre-generated part programs and comprehensive enough to use the CAD geometry of a workpiece to automatically generate the robot path, which may be translated into CNC part programs.

The results of preliminary testing (which will be presented in Chapter VI) led to a conclusion that the work presented in this thesis combined with the core functionality of Workspace5 software package are able to fulfil the requirements of the Flow International Corporation.

1.4. GENERAL LAYOUT OF THE THESIS

The thesis is organized as follows:

- Chapter I presents the thesis statement and explains how the dissertation is organized.
- Chapter II presents the background issues (concepts of: simulation, off-line programming, Realistic Robot Simulation interface, lexing and parsing, waterjet cutting technology, and G-code programming).
- Chapter III is the literature survey of different simulation and off-line programming software packages and compiler-compiler techniques.
- Chapter IV describes creation of the workpiece and 5-axis gantry robot CAD models and techniques for setting-up their kinematical properties.
- Chapter V encompases the robot language translator development, lexing and parsing of the native robot language, Realistic Robot Simulation interface calls embedded in Visual Basic for Applications programming language (i.e. creation of Workspace Simulation Language source code modules), implementation of the most important Realistic Robot Simulation services in the motion planner and the development of preprocessing graphical user interface.
- Chapter VI presents the development of off-line programming software module and the results of testing in the industrial environment.
- Chapter VII lists the conclusions of this research.
- Appendix A contains the survey of PRECCX basic features and programming techniques with appropriate examples.
- Appendix B is the glossary of parsing and lexical analysis key terms.
- Appendix C shows the G-code language and the CNC machine specification used for testing.
- Appendix D contains the mathematical equations used for mapping of CNC circular interpolation to robot circular interpolation.
- Appendix E contains the G-code grammar description using PRECCX syntax, with action code attached.
- Appendix F presents the set of C functions written for the parser customization.

- Appendix G shows the library of Workspace Simulation Language support functions.
- Appendix H contains the motion planner's C++ source code (Realistic Robot Simulation services are implemented in the motion planner module).
- Appendix I shows the C++ source code of the created off-line programming class.
- Appendix J shows the listing of the G-code part program generated off-line, based on the geometry of the testing part and robot path properties.
- List of references used and the Vita Auctoris are included at the end of the document.

CHAPTER II
BACKGROUND

In order to get the full understanding of the issues presented in the next chapters, basic concepts of simulation, off-line programming, lexing and parsing, Realistic Robot Simulation interface, waterjet cutting technology, and G-code programming language will be addressed in this chapter.

2.1. SIMULATION OVERVIEW

Simulation is the imitation of the operation of a real-world process or system over time. It involves the generation of an artificial history of the system and the observation of that artificial history to draw inferences concerning the operating characteristics of the real system that is represented (Banks, 1998). Simulation is an indispensable problem-solving methodology for the solution of many real-world problems. It is used to describe and analyze the behavior of a system, ask what-if questions about the real system, and aid in the design of real systems. Both existing and conceptual systems can be modeled with simulation.

The techniques of Computer Aided Design (CAD) have found extensive use in improving or replacing the process of engineering drawing, architectural drawing, and many other applications. However, an engineering process involving many moving parts can only be understood fully through the process of simulation.

Early developments in this field have involved the simulation of computer controlled machining centres using existing CAD systems. Additional software is used to take the original CAD drawing of a machined part and analyze a path across its surface. An animated simulation of the movements that a machining centre must go through to create the part from a "raw" block is then displayed. As well as providing a visualization of the process, a file can be created containing the required instructions to the machining centre. The file can then be executed to create the part.

This extension of computer aided design to computer aided manufacturing is highly applicable to industrial robotics. Simulation provides an efficient, interactive graphical environment which can improve the programming methods of industrial robots. Ever-increasing numbers of robot installations are now being planned using computer simulation. Only a few years ago the cost of the technology to achieve these requirements was prohibitive for all, but the largest organizations. However, as with CAD before it, simulation of industrial robots is possible even on standard, low-cost PC-compatible computers.

2.1.1. Object-Oriented Simulation

An object-oriented simulation models the behavior of interacting objects over time. Object collections, called classes, encapsulate the characteristics and functionality of common objects. A set of object classes has to be written in an object-orientated programming language (such as C++, Java, Smalltalk, etc.) by software developers, which are then used to create simulation models and simulation packages. The simulations built with these tools possess the benefits of an object-oriented design, including the use of encapsulation, inheritance, polymorphism, run-time binding, and parameterized typing. These concepts are illustrated by creating a set of objects to describe various simulation requirements. Object interactions define the behavior of any object-oriented simulation. In order to control the execution of the simulation, the development of a simulation language is mandatory, which has several notable features not available in other non-object-oriented simulation languages. Object-oriented simulations provide full accessibility to the base language, faster executions, portable models and executables, a multi-vendor implementation language, and a growing variety of complementary development tools.

The idea of an object-oriented simulation has great intuitive appeal because it is very easy to view the “real world” as being composed of objects. In a manufacturing cell, the physical objects may include machines, workers, parts, tools, conveyors, etc. However, part routings, schedules, work plans, and other information items could be also viewed as objects. All these objects interact to produce system behavior. A simulation engine manipulates these objects over the simulation run-time.

Since object-oriented simulations focus on objects, there is the possibility of dividing the simulation computation among objects. Objects provide a natural means of organizing the simulation and offer the potential of delegating portions of the execution to different processors, either parallel or distributed. Finally, since objects are often themselves made up of other objects, it is natural to decompose a system by its objects and view its behavior in terms of interacting objects (Banks, 1998).

2.1.2. Advantages and disadvantages of simulation

The benefits of simulation are mentioned in several technical references (*Waterjet Web Reference*, 2000), (*OMAX Abrasive Waterjets*, 2000), (*Flow International Coorporation*, 2000), and (Bo, 1994) which include the following:

1. Simulation lets one test every aspect of a proposed change or addition without committing resources to their acquisition. This is critical, because once the hard decisions have been made or the equipment has been installed, changes and corrections can be extremely expensive. Simulation allows one to test his designs without committing resources to acquisition.
2. By compressing or expanding time, simulation allows speed up or slow down phenomena to happen, so that important events can be investigated more thoroughly, while non important ones can be speeded up or skipped.
3. With simulation, one can determine the answer to the “why a certain phenomenon occurs" questions by reconstructing the scene and taking a microscopic examination of the system to find out the answers.
4. One of the greatest advantages of using simulation software is that once a valid simulation model have been developed, one can explore new policies, operating procedures, or methods without the expense and disruption of experimenting with the real system.
5. Simulation allows better understanding of the interactions among the variables that make up complex systems.
6. By using simulation to perform bottleneck analysis, one can discover the cause of the delays in work in process, information, materials, or other processes.
7. Simulation studies aid in providing understanding about how a system really operates

rather than indicating someone's predictions about how a system will operate.

8. Taking the designs beyond CAD drawings by using the animation features offered by many simulation packages the actual work of simulated facility or organization can be monitored. Therefore, it is possible to detect design flaws that appear credible when seen just on a two-dimensional CAD drawing.
9. The typical cost of a simulation study is substantially less than 1 % of the total amount being expended for the implementation of a design or redesign (Weisel, 1996). Since the cost of a change or modification to a system after installation is so great, simulation is in most cases a wise investment.
10. Simulation models can provide excellent training when designed for that purpose. The team, and individual members of the team, can learn by their mistakes and learn to operate better.

Conversely, simulation process is not ideal and flawless. According to Banks (1996), Law and Kelton (1991), and Pegden, Shannon, and Sadowski (1995) the main disadvantages of simulation include the following:

1. Model building requires special training. It is a skill that is learned over time and through experience.
2. Simulation results may be difficult to interpret. Since most simulation outputs are essentially random variables it may be hard to determine whether an observation is a result of system interrelationships or randomness.
3. Simulation modeling and analysis can be time consuming and expensive. Skimping on resources for modeling and analysis may result in a simulation model and/or analysis that is not sufficient to the task.
4. Simulation may be used inappropriately. Simulation is used in some cases when an analytical solution is possible, or even preferable.
5. Simulation is only as good as the model it is based on. Therefore, simulation cannot eliminate the errors made during the model design phase.

Potential users of simulation software packages should compare very carefully the benefits and the drawbacks of using the simulation to solve their industrial problems, because in some cases it might not be the most efficient engineering tool.

2.1.3. Cost of Simulation

An important fact that is often not considered is the cost to conduct computer simulation. A simulation study can easily cost as much as $30,000 just in staff time, (Owen, 1995). Average cost of a complete simulation that addresses all the user's concerns is estimated at $25,000-$100, 000, (Owen, 1995). The software itself runs from $2000 to $50,000, (Owen, 1995). The big costs are also in time-polling the people who know the system best and collecting the data the simulator needs.

2.2. OFF-LINE PROGRAMMING CONCEPT

Off-line or indirect programming refers to generating a robot program without interacting with a robot controller or using a programming device, remote from the industrial robot's workplace.

In contrast to this, so-called teach-in or direct programming occurs at the industrial robot's workplace by directly moving the end effector to the command positions, either with the help of a programming device or even manually, and then storing the positions in the robot controller's memory. The working positions of the robot are usually defined with sample workpieces (for instance sample car bodies). The use of this programming technique, which still dominates today, generally makes the halting of a production cell or a production line unavoidable during the set-up time and is therefore an important cost factor. The programming time in which the facility cannot be used productively may in some cases last days or even weeks. Therefore the most important reason for employing an off-line programming system is to reduce the set-up time.

A distinction has to be made between simple systems, for example textual programming systems, and costly off-line programming and simulation systems. The formers allow existing robot programs to be edited or archived, and are used to develop the program frame, usually without the determination of the individual positions and orientations of the robot motion. In combination with certain robot controllers textual programming systems also enable the syntax of robot programs to be checked. The latter costly off-line programming and simulation systems are interactively operated and allow a description to be made of the nominal path in the base coordinate system of the industrial robot. While textual programming systems are usually Personal Computer-based products, powerful workstations are often required for the three-dimensional real time graphics of the simulation systems.

Off-line programming systems still need to overcome several major problems to be widely accepted in automation industry. The following paragraphs will present the most important issues with respect to that.

The ideal case would be to load an off-line generated robot program down into the robot controller and to execute it without any adaptation. However this would assume good compatibility of the simulated and real manufacturing task. Even the leading companies in the domain of robot simulation and off-line programming, Technomatics and Deneb, do not claim that the simulation made by their software packages will completely match the reality every time their product is used (Nof, 1999). Primarily the reason for that is the discrepancy between the way a real robot controller and its virtual representation work. A possible remedy for that problem has been recently introduced in form of the Realistic Robot Simulation interface. The goal of the Realistic Robot Simulation is to use parts of the original controller software for more accurate simulation. The whole Realistic Robot Simulation concept is explained in detail in sections 2.3 and 5.2.2.1.

The development of postprocessors for translating from a simulation language to a robot language is not a straightforward task. The postprocessor must translate not only simple structures such as movement commands, but also more complex structures such as condition handlers, branching and looping statements, subroutines, and variables. There are very few established standards (Recommendations, rather than standards, have been developed by several national robotics associations. VDI (Verein Deutscher Ingenieure) robotics standard, developed in Germany, has had a certain success, but the problem was that only German manufacturers used it). The Realistic Robot Simulation standard is definitely the most accepted by robot manufacturers and certainly the most complete (Nof, 1999), and (Willnow et al., 1996).

Having carried out the translation, it is then necessary to transfer the program files from the computer running the simulation to the robot controller, either by using a communications link or else by copying the files onto a disk that is compatible with the controller. Again, this is an area in which there are few standards. Even the "standard" RS232 serial port poses problems of electrical wiring and protocol methods. However, once a link is established it is even possible to preprocess a robot program from its controller and translate into the language required by the simulation, allowing existing robot programs to be evaluated and improved.

2.3. INTRODUCTION TO REALISTIC ROBOT SIMULATION

In recent years, interactive and graphics-based tools for the planning, simulation and off-line programming of industrial robots have been introduced in industry. Compared to conventional planning and programming, Computer Aided Robotics (CAR) enables a better planning of robotic cells and reduces costly down-times of the manufacturing equipment due to on-line programming.

In order to depict robot manipulators, robot controllers, and task programs these Computer Aided Robotics software packages provide simulation models representing the real world equipment. The user applies these models via the user interface of the Computer Aided Robotics software package. Since different controller types possess different task languages, motion generator and methods for inverse kinematics, special controller models are required for an accurate simulation of each type.

Attempts to realistically model specific controller types have been undertaken by robotics specialists from the automotive industry and Computer Aided Robotics software suppliers. In general, these models showed a significant increase in simulation accuracy. However, without detailed controller knowledge, the implementation of specific controller models is very time-consuming. Furthermore, extensive measurement series are needed for the verification of the simulation model; yet this approach cannot guarantee the completeness and accuracy of the model. Whenever new controller type is introduced on the shop floor and its precise simulation is required, a new controller model must be implemented and verified.

As a consequence, these design flaws cause usually significant discrepancies between a robot's simulated and real behavior, generating the errors that are intolerable in today's high demanding industrial environment (RRS Maintenance Management, 1997). Refer to section 1.1 for more details.

The need to simulate more precise specific controller behavior has lead to the development of procedural interfaces within Computer Aided Robotics software packages. Today several Computer Aided Robotics software packages provide specific interfaces for the execution of controller models. By applying the interface documentation, robotics specialists can implement simulation models for dedicated controller or manipulators types. Depending on the Computer Aided Robotics software package, these interfaces allow different functionality and different data passing mechanisms. Therefore, a controller model provided for the interface a specific Computer Aided Robotics software package cannot be integrated in another Computer Aided Robotics software package without modification of the software.

To solve the above problems, the automotive industry has initiated the project "Realistic Robot Simulation" in which suppliers of robot controllers and robotic simulation systems cooperate. The project began on January 1, 1992 and was successfully completed in December 1993, (RRS Maintenance Management, 1997).

2.3.1. Objectives of the Project "Realistic Robot Simulation"

The project goal was to improve the simulation accuracy of industrial robot simulation systems in order to achieve a more realistic simulation of robot controllers.

The goal was achieved by defining a common software interface (Realistic Robot Simulation interface) that integrates the controller simulating modules into simulation software systems. Using original controller software parts, controller manufacturers provide simulation modules for their latest controller types. Simultaneously, simulation system suppliers have implemented the Realistic Robot Simulation interface in their software products (Figure 2.1).

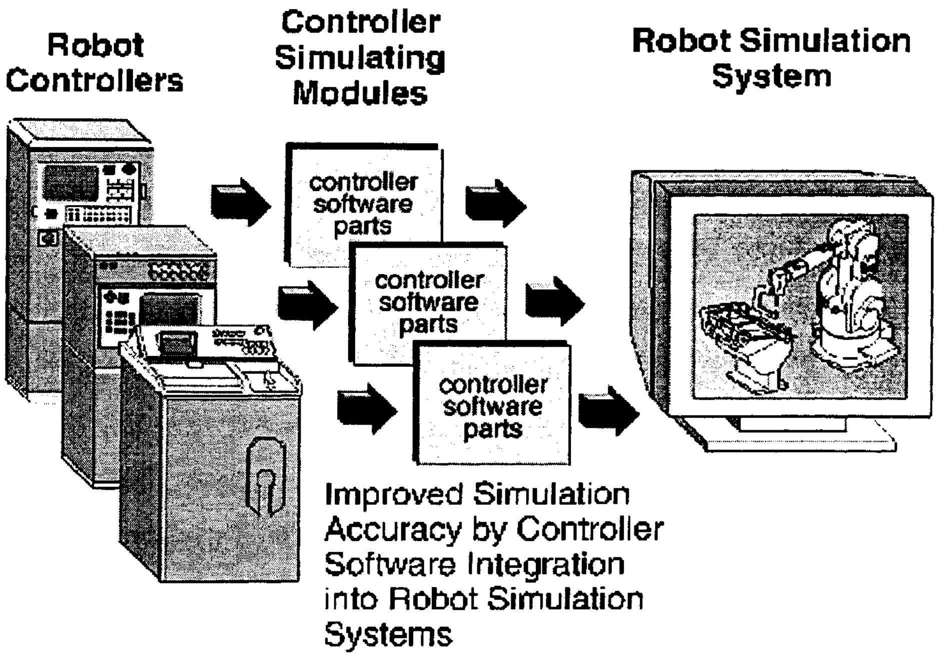

Robot Controllers
Controller Simulating Modules
Robot Simulation System
controller software parts
controller software parts
controller software parts
Improved Simulation Accuracy by Controller Software Integration into Robot Simulation Systems

Figure 2.1. Realistic Robot Simulation integration chart that shows the method to interface robot controllers and robot simulation software packages using Realistic Robot Simulation software modules.

2.3.2. Technical Aspects of the RRS Interface

In order to assure simulation accuracy and efficient implementation of original controller software, the Realistic Robot Simulation interface was derived from the robot manufacturer's controller software structures. The integrated controller software fulfils the requirements of the automotive industry for an accurate simulation of robot's:

- motion behavior,
- inverse and forward kinematics, and
- condition handling.

It has been proven that the deviation between simulated and real joint values is less than 0.001 radians, (Willnow et al., 1996). Concerning cycle times a difference of less than 3% could be reached (Willnow et al., 1996). The Realistic Robot Simulation interface runs on software and hardware platforms used for robotic simulation in the automotive industry.

2.3.3. Benefits and Disadvantages of the RRS Interface

There are several benefits of this interface:

- *Benefits of Robot Controller Suppliers*: By applying the Realistic Robot Simulation interface, controller suppliers can provide simulation products, which assure a better utilization of Computer Aided Robotics software packages at their customer's site. These simulation products can be used in all Computer Aided Robotics software packages equipped with this interface. Hence, by using this interface, controller suppliers do not need to implement *n* different simulation products for *n* Computer Aided Robotics software packages. Furthermore, they can focus on the development of dedicated products without reimplementing the general parts of Computer Aided Robotics software packages. Consequently, this effort allows the controller supplier to minimize implementation efforts.
- *Benefits of Computer Aided Robotics software suppliers*: Suppliers of Computer Aided Robotics software packages are no longer obliged to implement and verify specific controller models for accurate simulation of their Computer Aided Robotics software packages. Verification will become obsolete because the original controller software is used.
- *Benefits of Computer Aided Robotics software users*: By using original controller software within Computer Aided Robotics software packages, the simulation accuracy of the industrially applied Computer Aided Robotics software packages will be improved. This reduces costly downtimes of the manufacturing equipment. Once a new controller type is acquired, the automotive companies can also buy the corresponding simulation product for a precise simulation. Without a long implementation and verification phase, a precise simulation can be used during the initial operation phase of a new controller type, (RRS Maintenance Management, 1997).

The main disadvantage of the RRS interface is its cost. RRS software modules must be purchased directly from the robot manufacturers, and depending on the manufacturer, the module prices are in range from $5000 to $10000 (Willnow, 1996).

2.3.4. Availability of the RRS Interface

The Realistic Robot Simulation-Interface Specification has been available to the public (though only the companies that participated in the Realistic Robot Simulation project are entitled to obtain the Realistic Robot Simulation technical documentation) since January 1994. It is distributed by the Fraunhofer Robotics Institute in Berlin, Germany.

2.4. LEXING AND PARSING OVERVIEW

Lexers and parsers (definitions of these terms are provided below) help write programs that transform structured input. This includes an enormous range of applications; anything from a simple text search program that looks for patterns in its input file to a C compiler that transforms a source program into optimized object code (Levine, Mason, and Brown, 1995).

In programs with structured input, two tasks that occur repeatedly are dividing the input into meaningful units, and then discovering the relationship among the units. For a text search program, the units would probably be lines of text, with a distinction between lines that contain a match of the target string and lines that do not. For a C program, the units are variable names, constants, strings, operators, punctuation, and so forth. This division into units (which are usually called *tokens)* is known as *Lexical analysis,* or *Lexing* for short. Software programming utilities like Lex, or its dialect - Flex (so called *lexer generators*) take a set of descriptions of possible tokens and produce a C routine, which is called a *Lexical analyzer,* or a *Lexer,* or a *scanner* for short, that can identify those tokens. The set of descriptions given to a lexer is called a *Lex specification.*

The token descriptions that a lexer generator uses are known as *regular expressions.* A lexer generator turns these regular expressions into a form that the lexer can use to scan the input text extremely fast, independent of the number of expressions that it is trying to match. A lexer generated by a lexer generator is almost always faster than a lexer that might be written in C by hand, (Levine, Mason, and Brown, 1995).

As the input is divided into tokens, a program often needs to establish the relationship among the tokens. A C compiler needs to find the expressions, statements, declarations, blocks, and procedures in the program. This task is known as *parsing* and the list of rules that define the relationships that the program understands is a *grammar.* Software utilities such as PRECCX, YACC, Bison (so called *parser generators*) et cetera, take a concise description of a grammar, and produce a C routine that can parse that grammar - a *parser.* A parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar and also detects a syntax error whenever its input doesn't match any of the rules. A parser written by parser generator is generally not as fast as a parser that could be written by hand, but the ease in writing and modifying the parser is invariably worth any speed loss. The amount of time a program spends in a parser is rarely enough to be an issue, (Levine, Mason, and Brown, 1995).

2.5. INTRODUCTION TO WATERJET CUTTING

Waterjet cutting, also referred to as hydrodynamic machining, is an advanced technology characterized by a cutting tool that uses a stream of ultrahigh-pressure water forced through a sapphire nozzle. The water jet removes workpiece material and produces a narrow kerf (a cut in a workpiece made by a water jet stream) by the cutting action of a fine (.075 mm to .0.5 mm diameter), high velocity (more than twice the speed of the sound), high pressure (170 to 415 MPa) stream of water or water-based fluid with abrasives.

The first application of waterjet technology was in early 70s. Pressurized jet has been used in the timber, lumber, and pulpwood industries for many years as a means of quickly debarking logs from huge trees.

The concept of using an abrasive waterjet for machining metals was first developed in 1974, but since the mid-1980s practical equipment has become available for use only in precision machining. The principle was quite similar, the only difference was in the use of abrasive. Abrasive mixes with waterjet after the water flows out of the orifice, thus creating abrasive jet, which is much more efficient than the waterjet. Abrasivejet can cut almost any known material. Manufacturers of equipment now claim that the process can be used to cut "everything" from simple gray cast iron to 50 mm thick armor plate and boron-reinforced aluminum, (*Waterjet Web Reference*, 2000). Table 2.1. shows how cutting speed changes with respect to the type of material being cut (nominal thickness for all the samples is 50 mm).

Material	Cutting speed	
	[in/min]	[mm/min]
17-4PH Stainless steel	2.0	51
HY-80 High-strength steel	2.0	51
6A1-4V Titanium	2.0	51
Ni-Cr Superalloy	2.0	51
Aluminum	4.0	102
Lead	18.0	457
Glass	18.0	457

Table 2.1. The change of cutting speed with respect to the type of material being cut (nominal thickness for all the samples is 50 mm), (*Waterjet Web Reference*, 2000)

Figure 2.2 shows a relation between a pump pressure and the jet velocity during abrasive waterjet cutting.

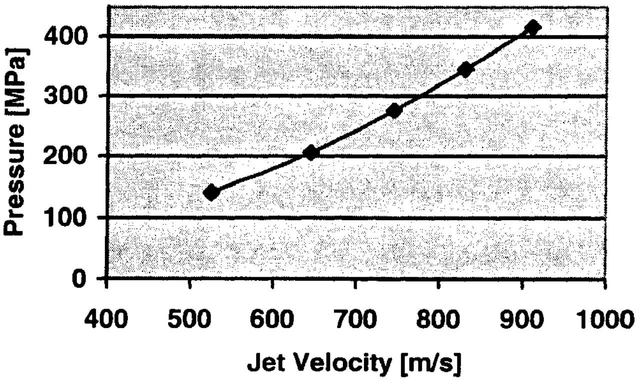


Figure 2.2. Relation between a pump pressure and the jet velocity during abrasive waterjet cutting, (*Waterjet Web Reference*, 2000).

2.5.1. Components of a waterjet cutting system

Waterjet and abrasivejet systems are almost the same, except for a few features. Main parts of both types of the systems are:

- filtering system,
- pump,
- manipulator,
- nozzle,
- control system, and
- catcher.

Abrasive jet has an additional abrasive recycling system, which collects the abrasive from a catcher, filters the reusable abrasive, washes and dries it, preparing it for the new cycle.

Filtering System

Water used in waterjet cutting has to be purified before entering the waterjet system. Inner diameters of the pipes used in waterjet systems are relatively small, and impurities can clog the system after long time of operation. Therefore the filtering system is the absolute requirement.

Water filtering can be done in several ways – through filtration, softening, or treatment by reverse osmosis.

Pumps

Waterjet and abrasivejet cutters use two types of pumps – intensifier pumps and crank pumps. Both types of pumps apply the same principle – the piston moves inside a cylinder, alternating directions, pressurizing, and pushing the water out of the cylinder into the waterjet system. However, there are many differences between the pumps:

- *The method of moving the piston* – crank pump moves the piston in exactly the same manner like an internal combustion engine. Intensifier pump uses a hydraulic cylinder to move the piston.

- *Energy conversion problem* – fluids are compressible under high and ultra high pressures. Under the pressure of 275 MPa, water is compressed for approximately 10%, (*Waterjet Technology*, 2000). The piston moves in one direction and raises the pressure, thus compressing the water. Once the outlet valve opens, water rushes out of the cylinder into the system. The pressure drops and the outlet valve closes. Water left in the cylinder expands, pushing the piston in the opposite direction. Energy resulting from water compressing and expanding is being treated differently:

1. *Crank pump* – converts this energy into the kinetic energy that moves the crank and other rotating elements, the same as the internal combustion engine,
2. *Intensifier pump* – energy has to be taken away by heat exchangers in order to provide the normal operation of the system. The result is different efficiency – crank pumps have efficiency of 95% and higher, while intensifier pumps have efficiency of approximately 70%, (*Waterjet Technology*, 2000).

- Pressure uniformity – intensifier pump usually pumps water in the system once or twice per second. Although the time interval is relatively short, still the pressure value jumps and drops enough to influence the cutting process significantly (uniform pressure is of vital importance for the cutting process). To eliminate pressure changes, an additional element in the pressure system is required – an accumulator. However, the problem with the accumulator is that it has to be massive in order to cope with ultra high pressures. Crank pump, on the other hand, makes around 30 strokes per second, thus the pressure remains uniform. Furthermore, there is no need for accumulator, (*Waterjet Technology*, 2000).
- Other differences

- *Noise* – crank pumps make less noise than intensifier pumps,
- *Maintenance costs* – crank pumps are easier to maintain, and parts are cheaper,
- *Price* – crank pumps are cheaper,
- *Operating speed* – operating speed relates to the piston speed. Crank pump moves the piston at speeds of approximately 0.75 meters per second, while the intensifier pump moves the piston at speed of approximately 0.15 meters per second, (*Waterjet Technology*, 2000). To achieve the same water flow, the intensifier pump must have

larger capacity, i.e. it has to be of larger size. This reflects in the higher price for the same water flow.

Selection of the optimal pump type is one of the most important tasks when it comes to introduction of waterjet cutting systems into industrial production. If the energy consumption represents a problematic issue, then a crank pump is a better solution. If the manufacturing process requires a lot of interruptions and a lot of changes, again the crank pump is the solution. However, for larger waterjet cutting systems, such as large gantry waterjet/abrasivejet cutters that work for long time without interruptions, intensifier pump is a better choice. Intensifier pump is a better solution for systems that use more than one nozzle, as well as the systems where the period between the preventive maintenance cycles must to be long.

Manipulators

The main purpose of the manipulators is to place a nozzle, i.e. a tool, in the desired position and orientation. Since most of the waterjet/abrasive cutting is done in an XY plane, manipulators are mostly of gantry type. The following pictures (Figures 2.3. and 2.4.) show some of the gantry type manipulators:

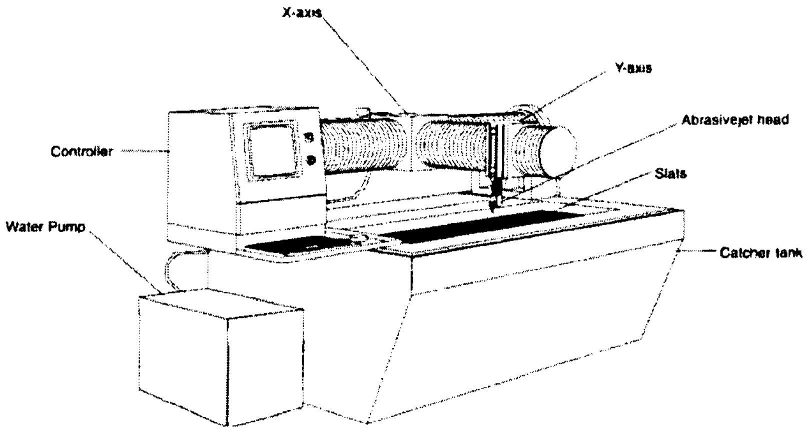


Figure 2.3. The sketch of OMAX 2-axes waterjet cutter, (*Omax Abrasive Waterjets*, 2000).

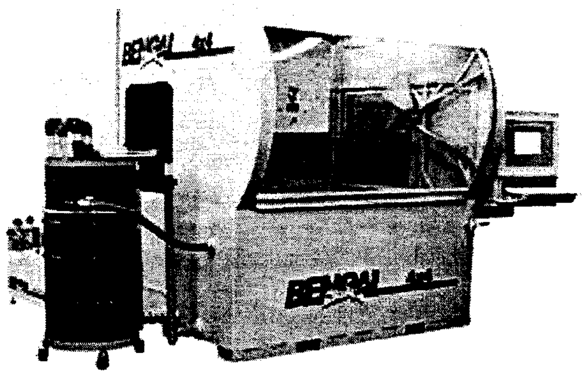


Figure 2.4. The photograph of Flow Automation 2-axes waterjet cutter, (*Flow International Corporation*, 2000).

Manipulators can be of different size and can be extremely large, such as Flow Corporation custom-made gantry manipulator that performs three-dimensional machining (Figure 2.5.). In this case, dimensions of the manipulator's work envelope are 20ft x 50ft x 5ft, (*Flow International Corporation*, 2000)

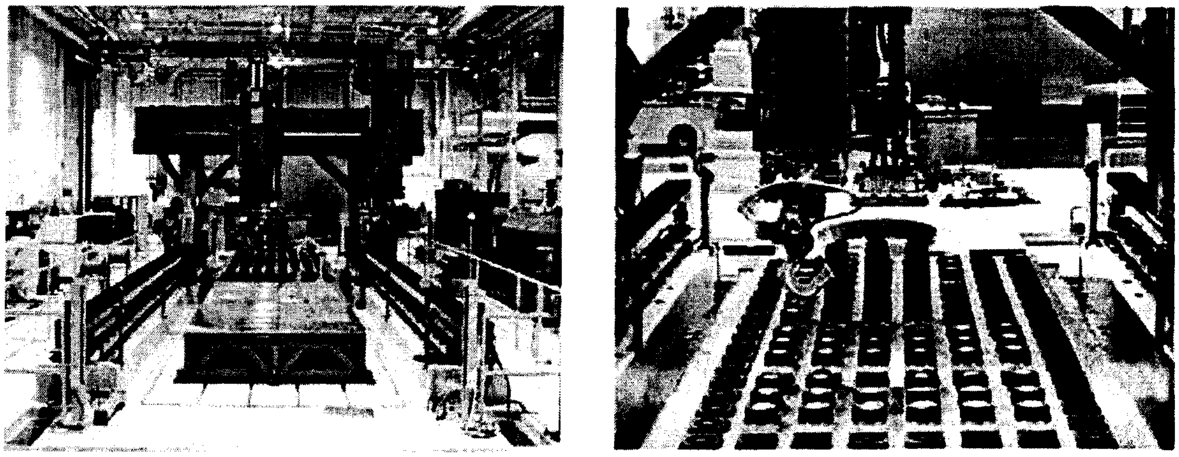


Figure 2.5. Two different views at Flow Automation 5-axes (3 translational and two rotational) gantry robot.

Other types of manipulators are robotic arm manipulators. For example, ABB I-R turnkey system represents a complete waterjet cutting cell system, which includes one or two inverted, vertically articulated robotic arms.

Nozzle

There are two types of nozzles used:

- waterjet nozzle, and
- abrasivejet nozzle.

The pictures below (Figure 2.6.) show both types of nozzles:

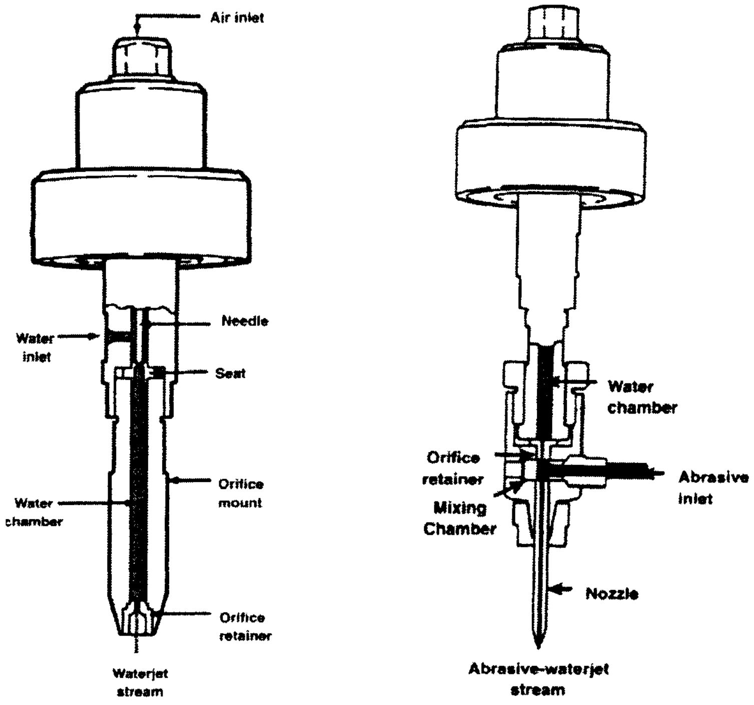


Figure 2.6. The cross-sections of a waterjet nozzle (left) and an abrasivejet nozzle (right), (*OMAX Abrasive Waterjets*, 2000).

Essentially, both types are the same, however the abrasivejet nozzle has an extra inlet that provides the flow of abrasive into the mixing tube. Simply by replacing a waterjet nozzle with an abrasive nozzle, the same machine becomes an abrasivejet cutter and vice versa. Replacement takes only several minutes, thus keeping the downtime to minimum, (*OMAX Abrasive Waterjets*, 2000).

Nominal life of a nozzle varies, and depends on the orifice quality and the seals inside the nozzle. The orifice is exposed to intense tearing, and seals are exposed to ultra high pressures. Usually, nozzle's life is approximately 40 - 75 hours, (Bo, 1994). Lately, a new type of nozzles, "long-life" nozzles were introduced on the market, whose lifetime ranges from 60 to 100 hours, (*OMAX Abrasive Waterjets*, 2000). Used nozzles do not have to be thrown away, they still can be used for rough cutting.

A key part of the nozzle is called a "jewel" or an orifice. The orifice is usually made out of sapphire, although other materials can be used instead. Diamond may be used as well, however the problem is its hardness, and the production of such a small orifice. Sapphire represents optimal solution, when it comes to quality, orifice life, and the manufacturing (*Flow International Corporation*, 2000). Another important parameter is the inner diameter and it ranges from 0.076 mm for soft materials up to 0.89 mm for hard materials, (*Waterjet Web Reference*, 2000).

It is important to note that the position of the orifice is located at the exit of the nozzle for the waterjet cutter, whereas for the abrasive cutter the orifice is located above the abrasive inlet. The reason is very simple. Abrasivejet involves more intense tearing than the waterjet, therefore the regular operational time of an orifice would be shorter if it was positioned passed the abrasive inlet.

In the waterjet nozzle, its position does not make any difference, since there is no abrasive in the water, and the abrasive effect is everywhere the same. Still, the orifice is located at the very end of the nozzle. If it were located farther from the nozzle end, the cutting efficiency would be partially lost due to the pressure distribution. Pressure significantly drops farther from the nozzle end, and that is the main reason for keeping the nozzle very close to the workpiece.

Control System

Control system represents the most important part of any waterjet/abrasivejet system. Basically, there are two types of the control systems:

- Manual control systems, and

- CNC control systems.

Before explaining the differences between the two types of control systems, the cutting process will be described. It is important to understand the process itself, because it will make the difference between the control systems more clear.

The most difficult problem of waterjet machining is the control of the cutting process. Although the water jet is a straight line "tool", it tends to "bend" while cutting a workpiece. Bending is used here to describe the shape of the jet. In case of two-dimesional machining, the point where the jet enters the workpiece and the point where the jet exits the workpiece should be on the same vertical line. This is not the case, however. There is always a lag that depends on several factors such as the nozzle speed, acceleration, material properties, and workpiece thickness.

The existence of the lag does not represent a problem if machining is performed along a straight line. There is always enough time for the jet to catch up. However, small radii and sharp changes in the direction are problematic. Since jet lags behind, it is not quite known where the jet will exit the workpiece. The only way to solve this problem, or at least to minimize its influence to acceptable level, is to decrease the speed every time the direction changes.

Manual control is used exclusively for trial and error testing. This kind of testing is used for determining the motion parameters that will provide the satisfactory cutting results. The number of tests that have to be performed is large, time consuming, and require highly skillful employees to conduct the tests.

On the other hand, each CNC controller manufacturer defines its own speed and acceleration profiles, which might be or might not be suitable for specific waterjet cutting operations. Still, the use of CNC controllers is a big step forward because of at least two reasons:

- G-code language typical for CNC controllers is simple and powerful language, and
- CNC controllers offer a lot of easy programmable features.

What is usually done in order to solve the jet lagging problem is to divide into segments all the lines and arcs of the cutter path, and then to set the motion parameters for each individual segment. However, there are at least two problems related to this solution. Firstly, the time required for segmentation, especially if the part geometry is complex, may be very long. The other problem is that the amount of memory required to store all the instructions can by far exceed contoller's buffer capacity.

Recycling System

Abrasive usage makes the machining cost higher, since the abrasive is one of the main components of the process. The costs of abrasive usage can built up to 50% of the total cost, (*OMAX Abrasive Waterjets*, 2000). Therefore it is very important to recover as much abrasive as possible in order to decrease the costs.

The way the recycling system is designed is very simple – sludge is pumped out of the water tank (catcher) into the recycling machine. The water entering the recycling system contains particles of abrasive and material the workpiece is made of. Water is filtered through the screens of different mesh size. Particles of the material as well as the abrasive of smaller size are deposited, while the rest of the abrasive is recovered. Recovery involves washing and drying of abrasive, thus preparing it for a new cycle.

Between 50 and 70 percent of the abrasive can be recycled after the first use, (*OMAX Abrasive Waterjets*, 2000). In turn, this decreases the waterjet cutting costs for up to 40%.

2.5.2. Advantages and disadvantages of waterjet cutting technology

The major advantages of waterjet cutting over other cutting techniques are:

- Application to flexible manufacturing systems, since changeover of cutting patterns is easily accomplished under computer control.
- The elimination of sharpening requirements.
- Increased production speeds.
- No heat-affected zone in the material being cut as with laser, electron beam and plasma arc cutting. Even abrasive waterjet cutting produces no heat that can degrade metallurgical properties, (*Waterjet Web Reference*, 2000). Because of the smooth edges produced, often no post machining is required.

- Maneuverability is also an excellent feature of this technology. Saws, for example, are mainly limited to straight cuts or essentially small arcs of circular cuts, whereas the jet can be easily directed to cut curves, holes, and complex shapes. In addition to metals and glass, waterjet cutting can be used to cut through large sections of concrete blocks.
- Material-savings through reduced kerf (a cut in a workpiece made by a water jet stream) and closer part spacing.
- Improved product characteristics through precise, clean edge cuts that eliminate crushing, hard edge, deformation, delamination, strings, burrs, or die slivers.
- Reduced dust, noise, heat, and sanitary problems.

Major disadvantage of waterjet cutting is its accuracy. Although accuracy has been improved with the new types of motion control, still the waterjet cutters have lower accuracy then EDM (Electro Discharge Machine) and laserjet cutting (two non-conventional machining processes similar to waterjet cutting) machines.

Laser machining achieves tolerances up to 0.025 mm and represents the most accurate of three methods, (*Waterjet Web Reference*, 2000).

Waterjet cutting achieves tolerance of up to 0.075 mm, (*Waterjet Web Reference*, 2000). The quality of the tolerances has improved significantly in last couple of years, due to the improved motion control.

EDM achieves tolerances that are in between the laser and waterjet cutting tolerances, (*Waterjet Web Reference*, 2000). Usually, waterjet cutting is used as first operation, used for quick cuts and rough tolerances, while EDM comes as second operation to bring the tolerances closer to the ones that were set.

This partially limits the application of waterjet cutting systems to areas where the high accuracy is not required (Waterjet cutting machines still cannot replace highly accurate milling and turning machining centers that can achieve machining tolerances of up to 0.001 mm). Waterjet cutting systems are often installed in the following industries:

- *Automotive Industry* – waterjet cutters have two areas of application. The first is a pre-processing role that involves the rough and quick cutting of a workpiece, followed by the fine cutting with EDM. The second area of application is important as well – cutting of doors, carpets, instrument panels, console parts, etc.

- *Aerospace Industry* – uses a lot of composite materials and alloys that are often difficult to be cut. Abrasivejet is the solution – it can cut almost any material with no heat generated. Cutting and drilling can be done without delaminating the material.
- *Food Industry* – water entering the waterjet system is absolutely clean of any kind of bacteria or microorganisms. Cutting is done in a very clean manner. There is no touch between the food and the nozzle, unlike with the knife. Cutting is done with no heat generated, so no potentially damaging processes can be started with waterjet cutting.
- *Tile and Marble Industry* – highly expensive diamond tools are replaced with the waterjet cutters. A case study showed that cutting speed increased almost 10 times, accuracy increased from 1.58 mm to 0.075 mm, and the control was significantly simplified, (Bo, 1994).

2.6. INTRODUCTION TO CNC CONTROLLER PROGRAMMING LANGUAGE (G-CODE)

A CNC controller performs machining operations by executing a series of commands that make up a part program. These commands are interpreted by the controller which then directs axis motion, spindle rotation, tool selection, and other CNC functions.

Part programs can be executed from the controller's memory or from a CNC tape (a memory device that can sequentially store CNC instructions) . Programs on tape can be executed directly from the tape, or can be loaded into the controller and executed from its memory.

Each machining operation performed by the controller is determined by the controller's interpretation of a group of words (commands) called "blocks" Individual blocks in a part program define each machining process. Part programs consist of a number of blocks that together define a complete operation on a part.

Part program blocks are made up of:

- **Characters** - A character is a number, a letter, or a symbol that has a specific meaning for the controller. For example, “1”, “G”, and “;” are characters that controller recognizes as meaningful information

- **Addresses** - An address is a letter that defines the instruction for the controller. The complete list of addresses for Allen-Bradley controllers is shown in the next table (Table 2.2.).

Function	Address
Rotary axis about X	A
Rotary axis about Y	B
Rotary axis about Z	C
Tool radius compensation number	D
Thread lead	E
Feedrate function (F word)	F
Preparatory function (G-code)	G
Tool length offset number	H
X arc centre in circular interpolation	I
Y arc centre in circular interpolation	J
Z arc centre in circular interpolation	K
Number of repetitions	L
Miscellaneous function	M
Sequence number	N
Program name	O
Subprogram name	P
Arc radius	R
Spindle rpm function	S
Tool selection function	T
Incremental axis name	U
Incremental axis name	V
Incremental axis name	W
Main axis	X
Main axis	Y
Main axis	Z

Table 2.2. List of programming addresses for Allen – Bradley 9 - series controllers

- **Words** - A word consists of an address followed by a numeric value. Examples of words are: G01, X15, F50, M2. For each word used in a part program, there is a format that designates the number of digits allowed as a numeric value for that word. The format for an M-code, which is a word, for example, is normally M2, which indicates that an M-address can be followed by only two digits.

- **Codes** – Represent CNC instructions that can be executed by CNC controllers. They are usually composed of words followed by parameters (in some cases a word alone can be a code. For example G17, which is a word, represents XY work plane selection, which is a CNC code at the same time). There are several groups of CNC codes: G-codes, which are motion and controller set-up instructions, M-codes which define program control and manufacturing application related instructions, T-codes which are tool related instructions, etc.
- **Parameters** - The controller has a number of fixed cycles that are initiated by a specific CNC code. When other words appear in those code blocks, they are referred to as "parameters", because their values are relevant only to that CNC code. For example, a Z word generally refers to a Z axis move, but when it appears in a block with a G83 peck drilling cycle, its value refers to the depth of the hole to be drilled. In that case, it is a "parameter" of the G83 fixed cycle.

A block is a set of codes that define the operations of the controller. For example:

/ N3 G00 X10 Z10 M3;

is a block composed of:

/ - optional block delete character,
N3 – sequence number word,
G00 – preparatory function word (rapid positioning mode),
X10 Z10 - axis movement words (parameters of G00),
M3 – miscellaneous function word (spindle on forward in clockwise direction), and
; - end of block character.

The controller sequentially executes blocks in a part program to conduct the required machining operation.

A part program has three logical sections:

- Beginning - setting up the controller and the machine to perform the operations wanted.

- Middle - performing the machining operations.
- End - returning the machine to a safe stop position, and preparing the controller for the next part program.

The blocks programmed vary for each section of the program. For example, the following simple program consists of the three mentioned logical sections.

```
G91G21;        -beginning

G00X28;
G33Z-46E4;     -middle
G00X5;

Z2;            -end
M02;
```

Example 2.1. Simple G-code part program divided into three logical sections

Furthermore, a complete part program may consist of a main program and subprograms, which can be called from the main program.

CHAPTER III

LITERATURE SURVEY

This chapter is divided in two subsections. The first subsection points out similarities and differences between the world's most utilized compiler-compiler (parser generator) software utilities YACC® and Bison®, and the one used in this study – PRECCX®, taking into account authors' opinions expressed in several technical articles and books in domain of parser generation. The aim is to show that PRECCX was chosen with solid reason, due to its indisputable advantages over YACC and Bison.

The second subsection presents the different simulation and off-line programming techniques introduced in score of technical articles. Those articles are divided into three different groups: the articles realated to development of language translators and off-line programming modules in robotics simulation software packages, the several general and introductory robotics simulation articles that have been mainly used to complement the simulation and off-line programming background sections of this dissertation, and finally the technical articles that describe the new techniques in simulation and off-line programming of industrial robots. The goal of this subsection is to present the work that has been done in the field of robotics simulation and off-line programming and to show how much this thesis is unique and in which aspects.

3.1 PARSER GENERATOR RELATED ARTICLES

Development of the robot language translator is one of the key parts of this thesis work. Compiler-compilers or parser generators are software utilities used to create various types of language translators (including translation of: one human-spoken language into another, a human-spoken language into a machine language, or a machine language into another machine language). Creation of fast, small (in memory size), and flexible (multi-purpose) translator primarily depends on the selection of a parser generator. Therefore, a considerable amount of literature research should be conducted before deciding which parser generator should be the most suitable for a translation task. Contemporary software market offers a variety of parser generators, but for majority of computer applications YACC® and Bison® parser generators are used. The reasons for the aforementioned are YACC and Bison's long-time availability on the market (since early eighties), several improved versions since the first release date, satisfactory amount of supporting documentation (insufficient or incomplete documentation is considered to be one of the major obstacles for less known parser generators to become more popular), and, of course, the variety of parsing options and techniques that these products offer.

Recently, a new parser generator has been introduced, named PRECCX (Prettier Compiler-Compiler Extended) offering several new features (such as: grammar contexts, synthesized attributes, infinite lookahead, complex expressions and macros) that can significantly shorten translation process, make the process of translation easier and more intuitive for developers, and generate parsers more modular and flexible.

This subsection presents the author's reasons for selecting PRECCX parser generator to be used in this dissertation, by pointing out PRECCX advantages and disadvantages over YACC and Bison. The technical articles written by Costagliola (1997), Rackovic (1996), Levine, Mason, and Brown (1995), and Breuer and Bowen (1992 - 1995) have been used as a knowledge base that has contained relevant information to justify the PRECCX parser generator selection. Thus, the next paragraphs sublimate the conclusions drawn after reading the mentioned articles. The definitions of the key terms required for good understanding of this section are presented in Appendix B.

PRECCX is intended to extend the Unix YACC and Bison utility. However, the technology is entirely different, which leads to some fundamental differences in the way that definition scripts have to be written. One can convert YACC or Bison scripts to PRECCX script quite easily, but PRECCX scripts cannot be converted to YACC or Bison scripts because of the extra expressiveness of the semantics involved. But the fundamental differences mean that sub-expressions cannot be converted independently of their context (YACC and Bison scripts are heavily context dependent), and some special features of YACC or Bison do not covert easily, such as precedence declarations, because they depend vitally on YACC or Bison semantics (Breuer and Bowen, *PRECCX User Manual*, 1999).

As far as possible, the PRECCX scripting language has been designed to look like an extension of YACC's and Bison's, with the result that PRECCX can be thought of as YACC or Bison with parameters, arbitrarily complex compound expressions, infinite lookahead and a better way of dealing with attributes.

The main advantages of PRECCX over YACC and Bison are presented below:

- **Contexts**

Each grammar definition may be parameterized with contexts. For example, *n* is the context in the following definition:

```
@       decl(n) = space(n) expression <'\n'> decl(n+1)*
```

This definition uses a grammar term *decl(n)*, which expresses the idea that a new line starts *n* spaces in from the left-hand margin. The right-hand side of the expression contains a term *decl(n+1)*, which designates that each following line will have one character longer indentation than the previous one. Some languages determine whether a declaration is local (and to what) or global in scope by relative indentation, and this is how to express this kind of constraint. It will be necessary to cast parameters to the type *PARAM (long)* if they are not of the same size (as *long*) under the model of C.

E.g.

```
@       decl1 = decl((PARAM)1)
```

This is rarely necessary.

- **Synthesized Attributes**

PRECCX can synthesize attributes immediately after matching a rule. An attribute is built by following the clause for which it is the attribute by an @, followed by the expression for the attribute, followed by a final @. The expression must not be side effecting, because PRECCX may execute the expression more than once if it backtracks. E.g.

```
@       foo = bar gum         {@ 1 @}
@       | nay                 {@ 2 @}
```

attaches the attribute 1 to the first clause and 2 to the second. Attributes already attached to the terms of the clause may be referenced and then dereferenced as follows:

```
@       arfarf = arf\x arf\y    {@ $x+$y @}
```

The dereferencing *$* in front of the *x* in *$x* is only necessary to ensure proper casting of types from (*PARAM* to *VALUE*) in all circumstances. It will usually not be required, but it is safer to use it. The *x* can and should always be used as a parameter without the *$*. E.g.

```
@       bowwow = bow\x wow(x)
```

This is where the real power of synthesized attributes comes in. An attribute synthesized during the parse can be used as a parameter in the remainder of the parse. This makes it possible, for example, to identify a single token:

```
@       foo = ?\x what(x)
```

whereas otherwise a construction like

```
@       foo = <'a'> what('a')
@       | <'b'> what('b')
@       | ...
```

would have been necessary. The attributes can be passed into actions too:

```
@       foo = ?\x       {: printf("%c",(int)$x); :}
```

but the actions are not executed until the end of the parse phase. In particular, it is no use expecting an action to alter an attribute value.

- **Infinite Lookahead**

PRECCX has infinite lookahead and backtracking in place of the YACC 1-token lookahead, This means that PRECCX parsers distinguish correctly between sentences of the form *foo bah gum* and *foo bah NAY* on a single pass. If one cannot imagine why he should want to decide between the two, a good example is to think about *if ... then* and *if ... then ... else*. One can write the grammar definition down straight away in PRECCX as

```
@       statement1 = <'i'> <'f'> boolexpr
@       <'t'> <'h'> <'e'> <'n'> statement
@       [ <'e'> <'l'> <'s'> <'e'> statement ]
```

but this is much harder to do for YACC-style.

- **Complex Expressions**

Complex compound expressions like

explain {{this | that} {several | no} times}+

are legal almost anywhere within PRECCX definition scripts. The definition can be substituted for the definee anywhere in a script except in the parameter list of a higher-order parser application. Grouping parentheses may be required.

- **Macros**

PRECCX *Macros* may be defined in a script, simply by defining one parser as a context for another. For example,

@ *optional(parser) = parser | {}*

may be defined (this particular example is an equivalent for the built-in [parser] construct). Then the construct

@ *ice_cream(flavour) = tub(flavour) optional(sauce)*

may be used. The *macros* are really ordinary grammar definitions, which just happen to take other grammars as parameters. It may be necessary to cast these parameters to be the same length as all the others, if the model of C uses different sized pointers for function addresses than *long*. The cast is only required when one introduces a grammar name as a constant:

@ *ice_cream(flavour) = tub(flavour) optional((PARAM)sauce)*

and he may also find that he has to declare

extern PARSER sauce;

somewhere above the line, just to let C know what is going on.

The main disadvantages of PRECCX in comparison with YACC and Bison and how important they are, are listed below:

- **Speed**

PRECCX compilation time is somewhat longer then in case of YACC and Bison. However, the compilation time is still short enough to be considered problematic. Therefore, PRECCX is moderately fast, typically taking two to five seconds to compile scripts of several hundred lines.

- **Action Execution**

There is a difference in PRECCX between the time at which the parse occurs and the time at which actions are executed. The parse occurs first and the actions are "built" during this phase, and executed either at the end of the parse, or at an explicit "!" command in the parse definition. This is in contrast to YACC and Bison where parse and the execution of actions are interleaved - but then PRECCX has to be able to backtrack across actions, and therefore cannot execute them immediately they are encountered. But, the complication of having to remember that the two phases are distinct is more than compensated for by the infinite lookahead that it allows.

- **Precedence**

There is at present no equivalent for the declaration of YACC or Bison precedences and associativity. Instead, these have to be coded explicitly for PRECCX using the preferred ordering.

It is obvious that PRECCX features marked as disadvantages are not even remotely as important for the parser to be generated as the features that are considered to be advantages over YACC and Bison. Therefore, according to conducted literature search, PRECCX has been considered to be the most appropriate solution for the task of robot language translation.

3.2. SIMULATION AND OFF-LINE PROGRAMMING RELATED TECHNICAL ARTICLES

The next several technical articles (below) have been reviewed in order to compare language translation process proposed by their authors with the translation process used in this thesis work.

Rackovic (1996) describes a method for construction and implementation of a robot language translator using compiler-compilers. Starting from the academic robot programming language PASRO, a new robot language is formed, as well as the translator for the newly formed language into the symbolic robot language of the robot control system EDUC-NET, using the compiler-compiler COCO-2. This paper presents very similar language translation methodology to the one applied in the dissertation. The major difference is that COCO-2 compiler-compiler lacks a lot of features that PRECCX has and therefore requires additional development of several supporting C files (very time consuming), and very long and sometimes trivial grammar rules (Trivial rules assign just a single token to the left hand side of the rule. They are mandatory in the grammars that can only accept symbols (left hand side of the rule) as parameters, but not single tokens.).

Kamisetty and McDermott (1992) concentrated their research on the design and development of the robot language translator for IBM SCARA (Selective Compliance Assembly Robotic Arm) configured robots. The translator converts simulation data from McDonnell Douglas' PLACE system into AML (A Manufacturing Language) robot language. This translator has been coded in C language. According to authors, the code maintenance is not straightforward, due to many function and variable dependencies. Also, grammar rules are coded in a programming language instead of scripting language, thus every rule change requires considerable changes in the source code.

All the developed software modules, which are the integral part of the thesis work, are written for the Workspace simulation platform. Also, if the software modules are analyzed separately, neglecting the interactions with other modules, the chances are that the wrong conclusions may be drawn. Therefore, to find out how efficient and accurate the concepts introduced in this dissertation are, the characteristics of whole platform along with added modules should be taken into consideration. The following literature survey has been conducted to compare such upgraded software platform with simulation and offline packages described in several technical articles. It is important to note that in this study Workspace has been compared to the software packages running on powerful workstations like IGRIP®, ROBCAD®, and SILMA®, which can offer considerably more to a user, but the price of those packages is more than ten times the price of Workspace. On the other hand, it will be proven that when compared to the other peer software packages presented in technical articles written by: Fujiuchi, et al. (1992), Danni, et al. (1996), Lee and ElMaraghy (1990), Rooks (1997), Krishnavrasad and McDermott (1992), Wozniak and Warczvnski (1989), Fukuda, Murakami and Kojima (1992), and, Zeghloul, Blanchard, and Ayrault (1997), Workspace has some strong advantages.

Due to the fact that there is no standard procedure that would define which criteria are the most significant for the simulation and off-line software package comparison, the twelve most common features that have been pointed out in almost every technical article regarding robot simulation, have been chosen to be comparison criteria (refer to the header row of Table 3.1). Some of the criteria are explained in more detail below:

- *Compatibility* – the ability of a software package to manipulate (export and import) different graphical file formats.
- *Customizability* – the ability of a software package to allow users an option to change the existing and to add new software features based on complexity of a simulated task.
- *Accuracy* – refers to how close a simulation software's interpretation of position and orientation of teach points, trajectory creation and cycle time calculation is to the one of a real robot controller.
- *Supported languages* – number of robot languages that can be verified and off-line programmed in a software package.
- *Automatic path* – refers to the ability of the software package to automatically create teach points using geometric features of CAD parts.

The other criteria are quite straightforward and do not need additional explanations. The inputs in Table 3.1 (below) are based on the facts provided in previously mentioned technical articles, product related technical documentation and personal experience formed by intensive analysis of four major robotics simulation packages (first four rows in Table 3.1) over the course of the last two years. The term "HIGH" that appears in the value section of the table is based on the benchmark set by the simulation package with superior performance based on a certain criterion. The terms "MEDIUM" and "LOW" denote gradually lower performances. This rating system is based on the personal experience and may possibly be subjective. The author has not been able to find any other, more objective simulation and off-line programming software package comparison in the literature.

According to the comparison shown in the Tables 3.1a and 3.1b, one can conclude that Workspace has better performance than the other Personal Computer based packages, but it still legs behind the most renown packages in robot simulation domain. However, Workspace has the highest performance/price ratio among all the packages in this analysis. (Nof, 1999)

	Compatibility	Customizability	Accuracy	Supported languages	Automatic path	Easy to use
IGRIP	HIGH	MEDIUM	HIGH	MANY	YES	NO
ROBCAD	HIGH	MEDIUM	HIGH	MANY	YES	NO
SILMA	HIGH	MEDIUM	HIGH	MANY	YES	NO
WS5	**MEDIUM**	**HIGH**	**HIGH**	**MANY**	**YES**	**MEDIUM**
ROBOSIM	MEDIUM	LOW	MEDIUM	A FEW	NO	YES
TOYSIM	LOW	LOW	HIGH	ONE	YES	YES
SMAR	HIGH	LOW	MEDIUM	N/A	NO	YES
SPOTS	LOW	LOW	LOW	MANY	NO	YES
SIMRO	LOW	LOW	MEDIUM	A FEW	NO	YES

Table 3.1. a. Comparative analysis of the basic features and characteristics offered in the examined simulation and off-line programming software packages.

	Dynamics Module	Statistical analysis	Robot models	Rendering	Price	Platform
IGRIP	YES	YES	MANY	HIGH QUALITY	HIGH	Work Station
ROBCAD	YES	YES	MANY	HIGH QUALITY	HIGH	Work Station
SILMA	YES	YES	MANY	HIGH QUALITY	HIGH	Work Station
WS5	**NO**	**YES**	**MANY**	**HIGH QUALITY**	**MEDIUM**	**PC**
ROBOSIM	NO	NO	A FEW	LOW QUALITY	N/A	PC
TOYSIM	NO	NO	A FEW	LOW QUALITY	MEDIUM	PC
SMAR	NO	NO	A FEW	LOW QUALITY	LOW	PC
SPOTS	NO	NO	A FEW	LOW QUALITY	LOW	PC
SIMRO	YES	YES	A FEW	LOW QUALITY	LOW	PC

Table 3.1. b. Comparative analysis of the basic features and characteristics offered in the examined simulation and off-line programming software packages.

Among all the criteria presented above, accuracy certainly determines whether a software package can be acceptable for robotics industrial applications. Off-line generated robot programs generally show considerable differences between the desired and the real motion after having been converted into robot controller's binary forms. Individual deviations include those from the nominal position and orientation of the end effector, from the nominal path and the path velocity, as well as from the nominal cycle time.

Investigations have been carried out in the last few years into increasing the simulation accuracy (Nof, 1999). With that respect, one of the conclusions is that if a robot's kinematics chain is more exactly modeled, position and orientation accuracy will be improved. Simulation packages that have accurate robot kinematics modeling ability enable more consideration to be given to manufacturing and assembly errors of the industrial robot axes and links, as well as, additional elasticity and the associated curvature.

In some other robot simulation packages, the transformation (inverse kinematics) and motion interpolation of the original robot controller software are integrated into the off-line programming system. It is thus possible to achieve improved path accuracy and other more realistic time-dependent characteristics, such as path accuracy, path velocity or cycle time. This accuracy issue is what distinguishes Workspace from the other Personal Computer based robotics simulation packages analyzed in this study. Workspace has a motion planner developed according to Realistic Robot Simulation specification, which can offer much more accurate simulation and therefore off-line programming to the user (This area is very important part of the dissertation. Detailed explanations are provided in sections 5.2.2.1 and 5.3).

3.3. INTRODUCTORY AND GENERAL PURPOSE ARTICLES

Articles written by Owen (1995), Owens (1991), and Weisel (1996) are very good starting point to acquire the basic concept of robotic simulation and off-line programming. They emphasize:

- The importance of simulation and off-line programming.
- Present the benefits for automation industry.
- Address the current problems and weak points.
- Foresee the future development in this area.

Because the first three points are fully elaborated in the dissertation, the next several paragraphs will introduce the expected near-future development in industrial robotics' simulation and off-line programming area, as presented in the articles above.

Best estimates suggest the size of the North American simulation market today is between $30 and $35 million (Nof, 1999). It is expected that in the next couple of years, the market will have surged to over $100 million, and some authors expect even exponential increase in the years to follow.

Within the next two to five years, the widespread entry of Personal Computer-based open architecture controllers will be in demand. They will replace today's specialized controllers that require specialized interfaces. The leading Personal Computer simulation software packages will be compatible with most, if not all, of those Personal Computer controllers. Personal Computer-based, resident simulation in engineering at vendor facilities, builder's shops and at production facilities will be mandatory.

Simulation will be neutral to the robot and specific to the application. Systems integrators and robot users will demand objective analyses of the entire range of robot manufacturing, and robot manufacturers therefore will not be in the field of objective simulation analysis of robotic systems.

Factories with robots from different manufacturers will increasingly rely on simulation and demand services from companies that do not have ties to robot or supplier companies. Users will insist on an objective evaluation of robot function and system performance for a wide range of manufacturers' products.

New techniques of simulation and off-line programming will be developed. Virtual reality and knowledge-based systems will be used to facilitate the interactions between a robot and a user, and also to make the process faster, more "natural" and more precise. The next section presents the theoretical outline for those two techniques.

3.4. NEW TECHNIQUES IN SIMULATION AND OFF-LINE PROGRAMMING OF INDUSTRIAL ROBOTS

3.4.1. Virtual Reality Approach in Simulation and Off-line Programming of Industrial Robots, (Boud and Steiner, 1999)

This technical article discusses the development of a new method for the off-line programming of robotic devices, and also indicates some of the potential applications.

In comparison with Workspace (off-line programming software package used as a software tool in this thesis), off-line programming within a virtual environment could reduce the required skill levels of a programmer, reduce the programming times, allow the operator a "natural" interface with which the operator would conduct the task in the real world, and reduce the monotony.

The use of Virtual Reality as an off-line programming technique is a natural evolution of simulation packages already available for the robotics, including Workspace.

The major areas where Virtual Reality technology can be applied to the robotic applications can be listed as follows:

- *Simulation of manufacturing plants, and planning of robotic workcells.* This can give the operator a sense of "being there", interacting with the virtual environment. By using shuttering glasses, the user can obtain a perception of depth, which would not otherwise be available in a two-dimensional format. It is as close to three-dimensional perception as possible.
- *Off-line robot programming from manipulations carried out by the "natural" movements of the operator.* This follows on from a similar principle previously known as a "robot training arm". This required the robot to be stopped during production in order to be programmed on-line. However, using for example an instrumented glove as part of a Virtual Reality system, the operator's gestures are fully captured and hence the user's movements are deemed to be more "natural".
- *Teleoperation of robots in remote places.* Applications in this area include the control of robots in hazardous environments or in distant locations (such as space exploration) where operations are extremely expensive both to plan and to execute.

Advantages of using Virtual Reality based systems over Workspace are as follows:

- *Immersibility.* The presentation of pictures in a three-dimensional view. An operator visualizes a three-dimensional world with the viewpoint changing interactively when the user moves his head. An illusion of the operator being inside is therefore obtained. Workspace has dynamic zoom and dynamic orbit functions that cannot achieve such a high level of three-dimensional representation.
- *Interactivity.* Advanced input devices such as DataGloves and six degree of freedom Joysticks enable the operator to manipulate interactively within the virtual environment. Workspace uses standard input devices - mouse and keyboard, which are not that convenient.
- *Dynamics.* Virtual Reality systems are characterized by a high arithmetic and graphical performance. Immersibility and interactivity need a system that can react

rapidly to the inputs of the operator. This guarantees a continuous simulation of dynamic processes. Workspace has simulation engine under development, still suffering of some functional errors. Also, Workspace still cannot play real-time simulations, although the real time/simulated ratio is mostly acceptable (between 1:2 and 1:3).

An initial Virtual Reality system is proposed by the authors (Figure 3.1.), and is under consideration for development at the University of Birmingham by the Intelligent Systems and Robotics research Group.

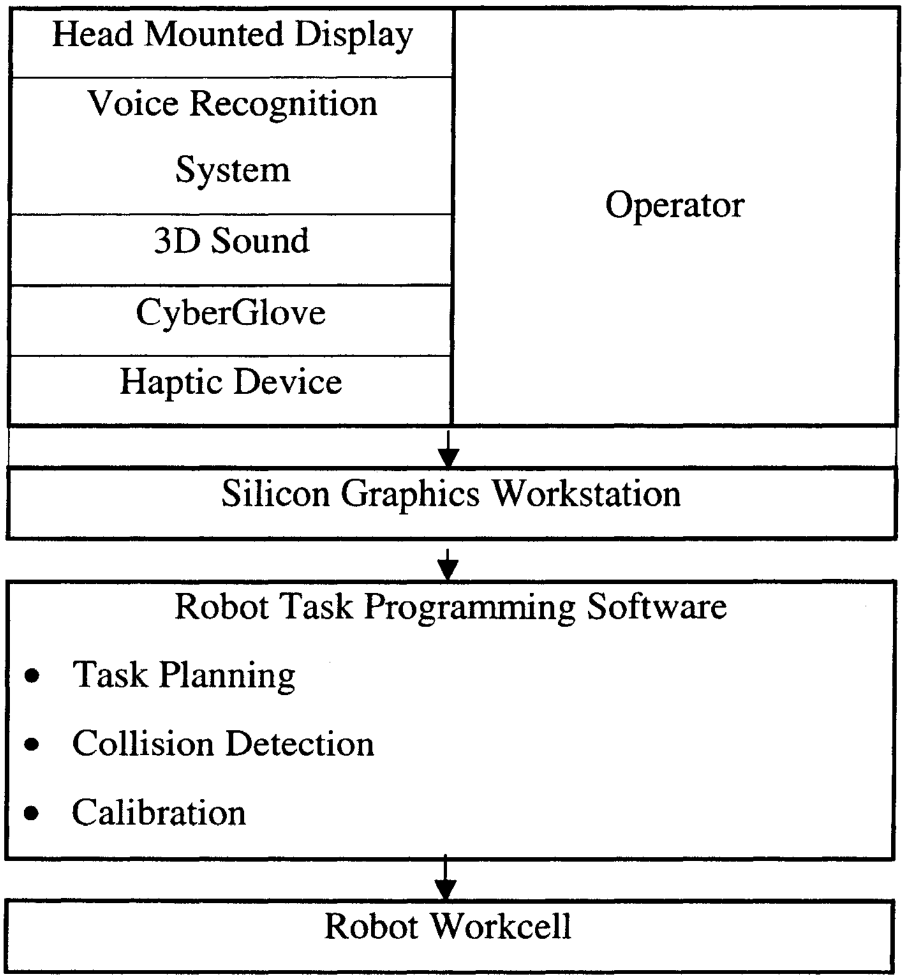


Figure 3.1. Initial proposal of the off-line robot programming facility being developed at the University of Birmingham.

3.4.2. Knowledge Based Simulation and Off-line Programming Approach, (Bernhard, Schahn, and Schreck, 1999)

The knowledge based programming procedure is composed of two phases (Figure 3.2.).

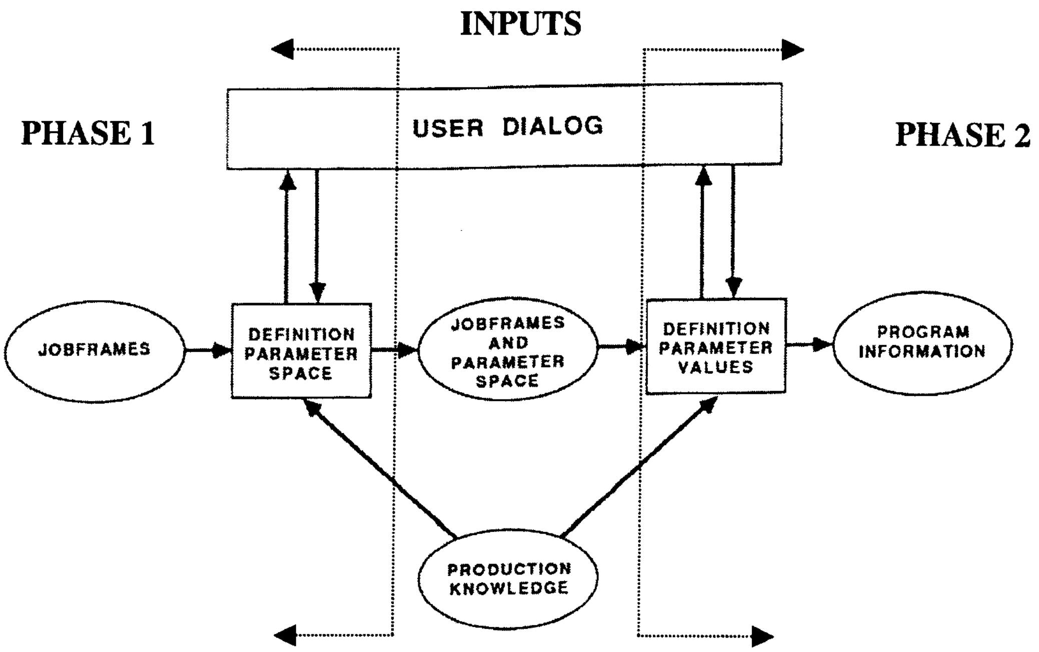

INPUTS
PHASE 1
USER DIALOG
PHASE 2
JOBFRAMES
DEFINITION PARAMETER SPACE
JOBFRAMES AND PARAMETER SPACE
DEFINITION PARAMETER VALUES
PROGRAM INFORMATION
PRODUCTION KNOWLEDGE

Figure 3.2. Principal knowledge based approach proposed by the authors.

In the first phase the required production task specific parameter space is determined. In the second phase the parameter values are defined.

Based on the geometric task information described by the jobframes and the actual state of the production cell, the program information can be in principle derived from the rules representing detailed production knowledge.

The principal system architecture is shown (Figure 3.3.) which is prototypically realized. The main elements of this architecture are:

1. Production cell components,
2. Static cell model,
3. Dynamic cell model,
4. Production knowledge, and
5. Control strategies.

All the elements are described subsequently.

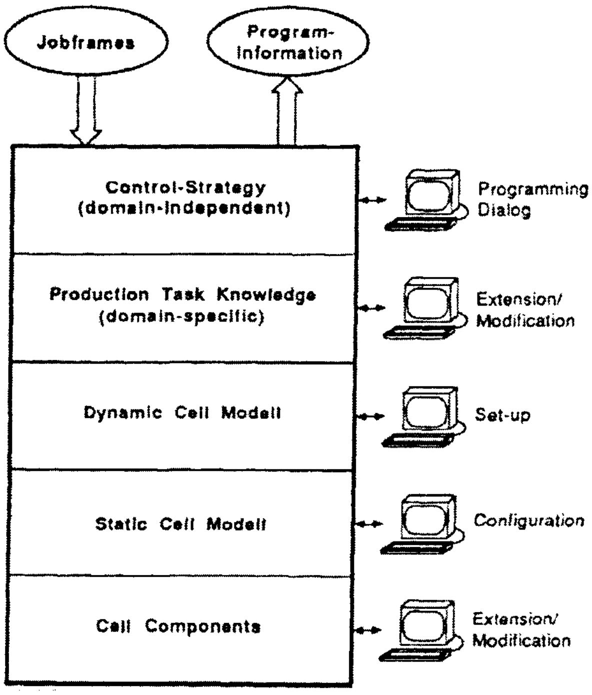


Figure 3.3. Prototypically realized system architecture

The production cell components (such as: robots, robot controllers, workpieces, and tools) are the basis for cell modeling. According to the authors, frames seem to be the most appropriate solution for modeling the components. Frames are comparable with named entities consisting of slot-value pairs. Thereby the entities characterize objects and the slots their attributes, which may have values. The frames may be structured in classes, sub-classes and instances of classes. Figure 3.4. presents the frame representation of a welding gun.

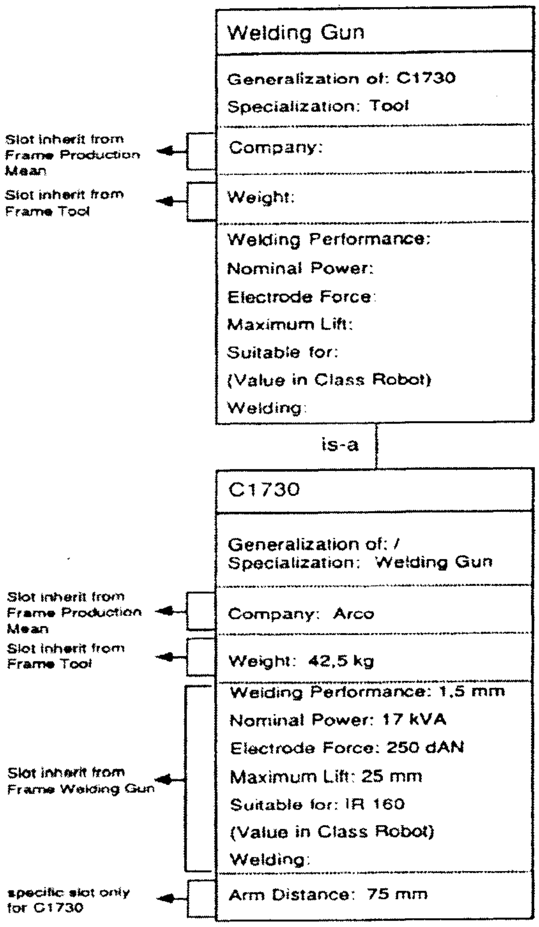


Figure 3.4. Frame representation of welding guns.

The static production cell model is represented by the fixed configuration of selected cell components. Similar to components, each production cell is described by a frame, which can be associated with the cell components by the relation “contains”, defined as a slot in the frame.

The dynamic cell model reflects the states of the components and their interrelations within a configured cell. The interactions with other components are represented by special relations, determined in the static cell model. In dynamic cell model methods and rules are added in order to change the states with regard to the connections among the components.

The next important area for the realization of the system is related to the acquisition and representation of the production knowledge, required to infer the program information. It can be subdivided into acquisition and representation of general valid facts, e.g. material constants and specific, mostly incomplete facts especially regarding jobframes, the cell model and general facts related to the program information. The first area of production knowledge is mainly represented by frames, while for the latter one, a rule representation was used.

To get the system working, additional procedures are required. They are realized by introducing the control strategies, which define the actions of the system independent of the production specific knowledge and the modeled cell.

To derive the parameter values the system tries to infer a value by examining every possible rule chain without asking the user. If a complete chain to infer the parameter value cannot be found, the system asks the user to provide the value.

The main similarity between presented system and Workspace is that the Program Information in the proposed system and geometric points in Workspace are virtually the same. Program Information is composed of Jobframes, Motion Information, Sequence Information, Process Parameter and Cell Information. The same items, just named differently, are contained in *GP Properties* dialog box in Workspace.

The main difference is related to user interaction. While in Workspace user needs to completely define all the parameters important to the application that will be simulated, the presented system does that automatically. The system tries to infer a parameter value by examining every possible rule chain without asking the user. If a complete chain to infer the parameter value cannot be found, the system then asks for the user's input. The system has a knowledge base, so the process parameters are chosen to optimize the work path, thus eliminating user's input errors. The corresponding benefit is that the user does not need to know a lot about the process that he wants to simulate, the system will do most of the "thinking". On the other hand, Workspace, in case of undefined parameters, uses default values that may or may not be appropriate for a simulated process.

CHAPTER IV

ROBOT MODELING AND KINEMATICS

This chapter contains the following sub-topics:

- The architecture of mechanical manipulators.
- The major tasks of robot kinematics.
- The review of the CAD based graphic simulators for robotic systems.
- The list of software modules that comprises a typical robotics simulator.
- The description of the workpiece and the 5-axis gantry robot CAD and kinematics design techniques used in this work.

4.1. ABOUT ROBOT MODELING AND FUNDAMENTAL QUESTIONS OF ROBOT KINEMATICS

A mechanical manipulator can be modeled as an open-loop articulated chain with several rigid bodies (links) connected in series by either revolute or prismatic joints driven by actuators. One end of the chain is attached to a supporting base while the other end is free and attached with a tool (the end-effector) to manipulate objects or perform assembly tasks. The relative motion of the joints results in the motion of the links that positions the hand in a desired orientation. In most robotic applications, one is interested in the spatial description of the end-effector of the manipulator with respect to a fixed reference coordinate system.

Robot arm kinematics deals with the analytical study of the geometry of motion of a robot arm with respect to a fixed reference coordinate system as a function of time disregarding forces and moments that cause the motion. Thus, it deals with the analytical description of the spatial displacement of the robot as a function of time, in particular, the relations between the joint-variable space and the position and orientation of the end-effector of a robot arm. This section addresses two fundamental questions of both theoretical and practical interest in robot arm kinematics:

1. For a given manipulator, given the joint angle vector $q(t) = (q_1 (t), q_2 (t), \ldots q_n(t))^T$ and the geometric link parameters, where *n* is the number of degrees of freedom, what is the position and orientation of the end-effector of the manipulator with respect to a reference coordinate system?
2. Given a desired position and orientation of the end-effector of the manipulator and the geometric link parameters with respect to a reference coordinate system, can the manipulator reach the desired prescribed manipulator hand position and orientation? And if it can, how many different manipulator configurations will satisfy the same condition?

The first question is usually referred to as the *direct (or forward) kinematics* problem, while the second question is the *inverse kinematics (or arm solution)* problem.

Since the independent variables in a robot arm are the joint variables and a task is usually stated in terms of the reference coordinate frame, the inverse kinematics problem is used more frequently.

Since the links of a robot arm may rotate and/or translate with respect to a reference coordinate frame, the total spatial displacement of the end-effector is due to the angular rotations and linear translations of the links. Denavit and Hartenberg (1965) proposed a systematic and generalized approach of utilizing matrix algebra to describe and represent the spatial geometry of the links of a robot arm with respect to a fixed reference frame. This method uses a 4 x 4 (four rows and four columns) homogeneous transformation matrix to describe the spatial relationship between two adjacent rigid mechanical links and reduces the direct kinematics problem to finding an equivalent matrix that relates the spatial displacement of the "hand coordinate frame" to the reference coordinate frame. These homogeneous transformation matrices are also useful in deriving the dynamic equations of motion of a robot arm.

In general, the inverse kinematics problem can be solved by several techniques. Most commonly used methods are the matrix algebraic, iterative, or geometric approaches.

4.1.1. CAD based graphic simulators for robotic systems

CAD-based graphic simulators for robotic systems allow CAD modeling of robot manipulators and facilitate the solutions for forward and inverse kinematics problems. Cell design, robot selection, and verification of the robot reach and of the correct placement of the cell elements, can all be done in a virtual CAD and simulation environment.

The main elements of a CAD based graphic simulation system are (Nof, 1999):

- A CAD solid modeler, which allows the user to build a database with a valid and complete geometric description of the robot and its environment. This CAD database includes the models of the robot links and all the objects in the cell environment: machines, fixtures, feeders, grippers, parts, etc.
- Built-in libraries of commercially available industrial robots, common production equipment, and application-specific options that include tooling, such as a variety of commercially available grippers, spot weld guns, and arc weld guns. The libraries are an integrated part of the system and can be expanded by inclusion of elements from the CAD database created by the user.
- Data translators for standard data-exchange formats such as IGES, DXF, and STEP. These translators allow the importing of models of products, tools, and parts from other corporate CAD systems and support the rapid development of accurate simulation models. They also allow the exporting of model data to be used by other systems.
- Kinematics module, which allows the modeling of the robots and other mechanisms. This module includes direct and inverse kinematics algorithms, which are necessary to calculate the robot envelope, its reach, and the motion in space during simulation of the robot's movements.

The various commercially available robotic simulation packages provide the basic tools needed for robotic cells and system design. These basic tools are essential in order to rapidly design and deploy automated manufacturing systems. A software package called Workspace has been developed as a visualization tool for engineers and managers involved in the process of designing or debugging new or existing robot installations. In this dissertation Workspace5 software package has been used as a CAD graphics and robot kinematics modeling platform.

4.2. THE DESCRIPTION OF ROBOTIC CELL DESIGN PROCEDURE USED IN THE DISSERTATION

The following step-by-step procedure has been used to completely define all the elements of the robotic cell, designed for simulation and off-line programming testing: the manipulator, the workpiece and the cutting tool. The manipulator has beem modelled according to the technical specification of an AF-Series 5-Axis Three Dimensional Shapecutting Machine, manufactured by Flow Robotics Company. Similarly, the CAD model of a cutting tool has been created based on PASER3 waterjet cutting tool specification. The tool has been manufactured by Flow Automation Company.

The procedure is as follows:

- Create a CAD solid model of the robot.
- Rename the robot CAD components to have the Workspace required names. For the robot base use the robot model's name, and for the robot links the following names should be used: Link1, Link2, Link3, Link4, Link5 (Figure 4.1).

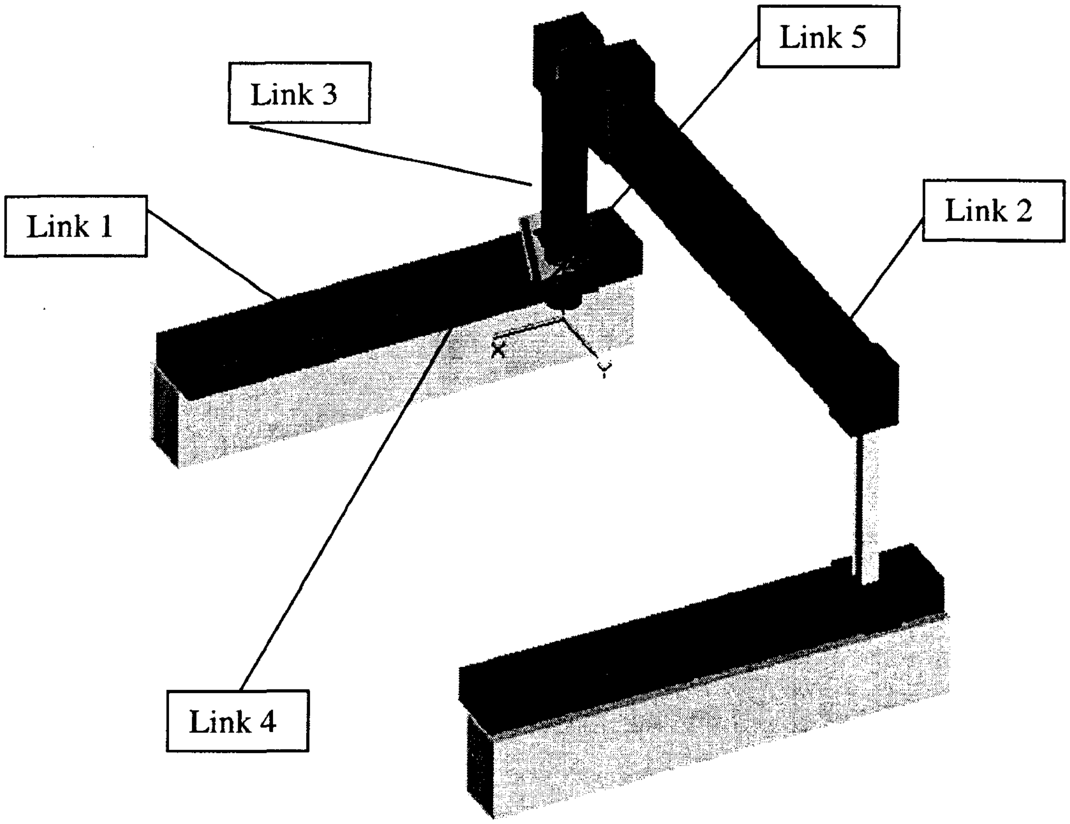
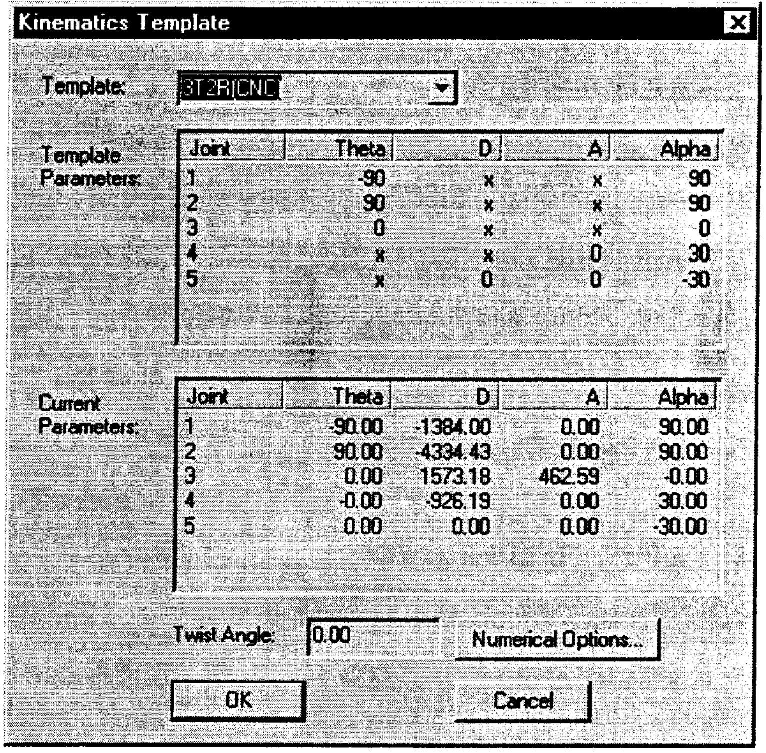


Figure 4.1. CAD structure of the manipulator composed of 5 links connected at joints.

- The next construction stage is to join the links into a robot structure. For this operation co-ordinate frames must be placed at the joint centres and oriented correctly.
- Select the base of the robot and use the menu options to create the robot.
- Each joint must be in the correct position. To change their positions, use appropriate menu options.
- Define the matching kinematics template by entering the values of kinematics parameters in Denavit-Hartenberg matrix template and associate it with the CAD object of the manipulator (Figure 4.2).



Template: 3T2R|CNC

Template Parameters:

Joint	Theta	D	A	Alpha
1	-90	x	x	90
2	90	x	x	90
3	0	x	x	0
4	x	x	0	30
5	x	0	0	-30

Current Parameters:

Joint	Theta	D	A	Alpha
1	-90.00	-1384.00	0.00	90.00
2	90.00	-4334.43	0.00	90.00
3	0.00	1573.18	462.59	-0.00
4	-0.00	-926.19	0.00	30.00
5	0.00	0.00	0.00	-30.00

Twist Angle: 0.00

Figure 4.2. Inverse and forward kinematics template that shows the parameters of the Denavit-Hartenberg's matrix

- Add the World Coordinate Frame to the robot.
- Add the Tool Coordinate Frame to the robot (Figure 4.3).

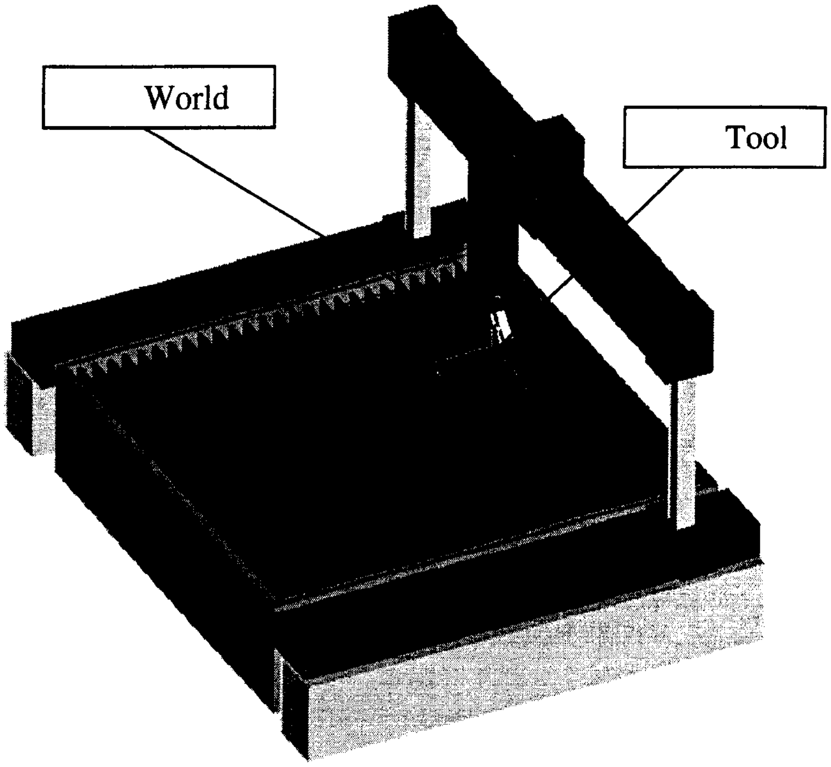
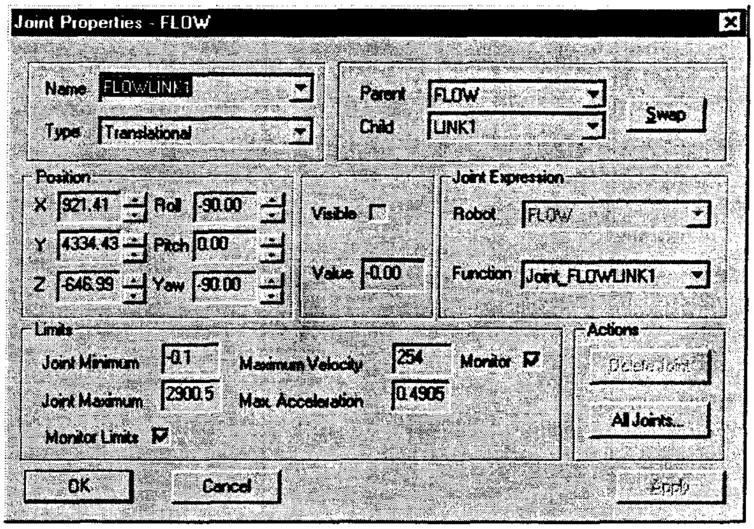


Figure 4.3. Position and orientation of the Tool and World coordinate frames attached to the CAD model of the manipulator

- Enter the robot limits for each joint (Figure 4.4).
- Enter the robot Velocity and Acceleration values (Figure 4.4).



Joint Properties - FLOW

Name: FLOWLINK1
Type: Translational
Parent: FLOW
Child: LINK1

Position
X: 921.41 Roll: -90.00
Y: 4334.43 Pitch: 0.00
Z: -646.99 Yaw: -90.00

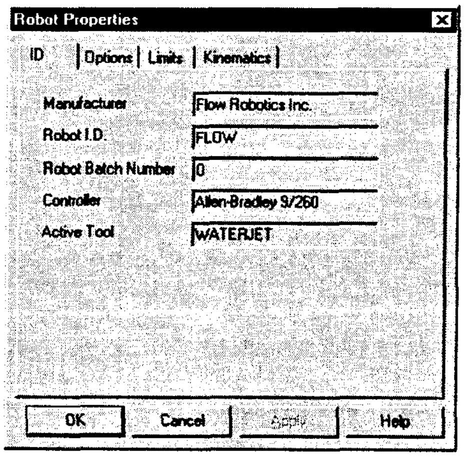
Visible □
Value: -0.00

Joint Expression
Robot: FLOW
Function: Joint_FLOWLINK1

Limits
Joint Minimum: -0.1 Maximum Velocity: 254 Monitor ☑
Joint Maximum: 2900.5 Max. Acceleration: 0.4905
Monitor Limits ☑

Figure 4.4. Robot joint properties dialog box that defines joint positions, joint limits, joint velocities and accelerations, and joint connections within a kinematic chain.

- Check the robot joint motions for all the possible robot link collisions. If a collision is detected, modify the model so that the robot is not allowed to collide with itself. Robot's collisions with the simulation environment entities (other robots, mechanisms, turntables, conveyors, etc.) Workspace detects automatically during the simulation run time.
- Define the Home position of the robot.
- Define the Zero position of the robot.
- When the robot model is complete, define robot general properties (See Figure 4.5).



Robot Properties

Manufacturer: Flow Robotics Inc.
Robot I.D.: FLOW
Robot Batch Number: 0
Controller: Allen-Bradley 9/260
Active Tool: WATERJET

Figure 4.5. Robot properties dialog box that provides general information about the manipulator and its controller.

- Check the robot-working envelope. The working envelope is important to show the limits of robot's operational space and also for safety considerations (Figure 4.6).

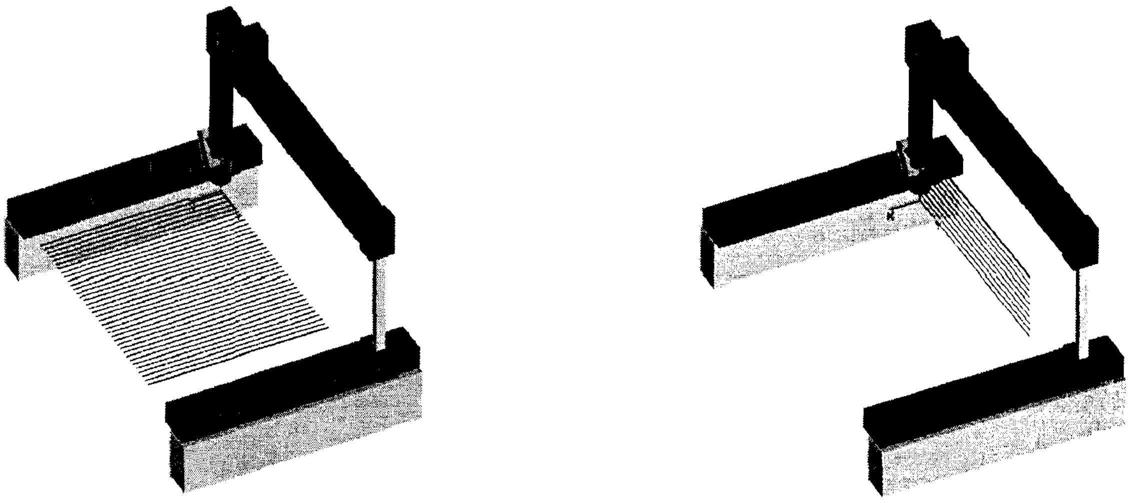


Figure 4.6. Robot's working envelope created separately for axes 1 and 2.

- Design the CAD model of the workpiece that will be complex enough to require 5-axis machining. Designing a workpiece that would require 3 or 4-axis machining is possible of course, but the complex 5-axis robot kinematics wouldn't be fully tested in that case. Therefore, a wheel-shaped workpiece has been modeled, with the inner contour that requires synchronous cutting motion of all five robot axes (Figure 4.7).

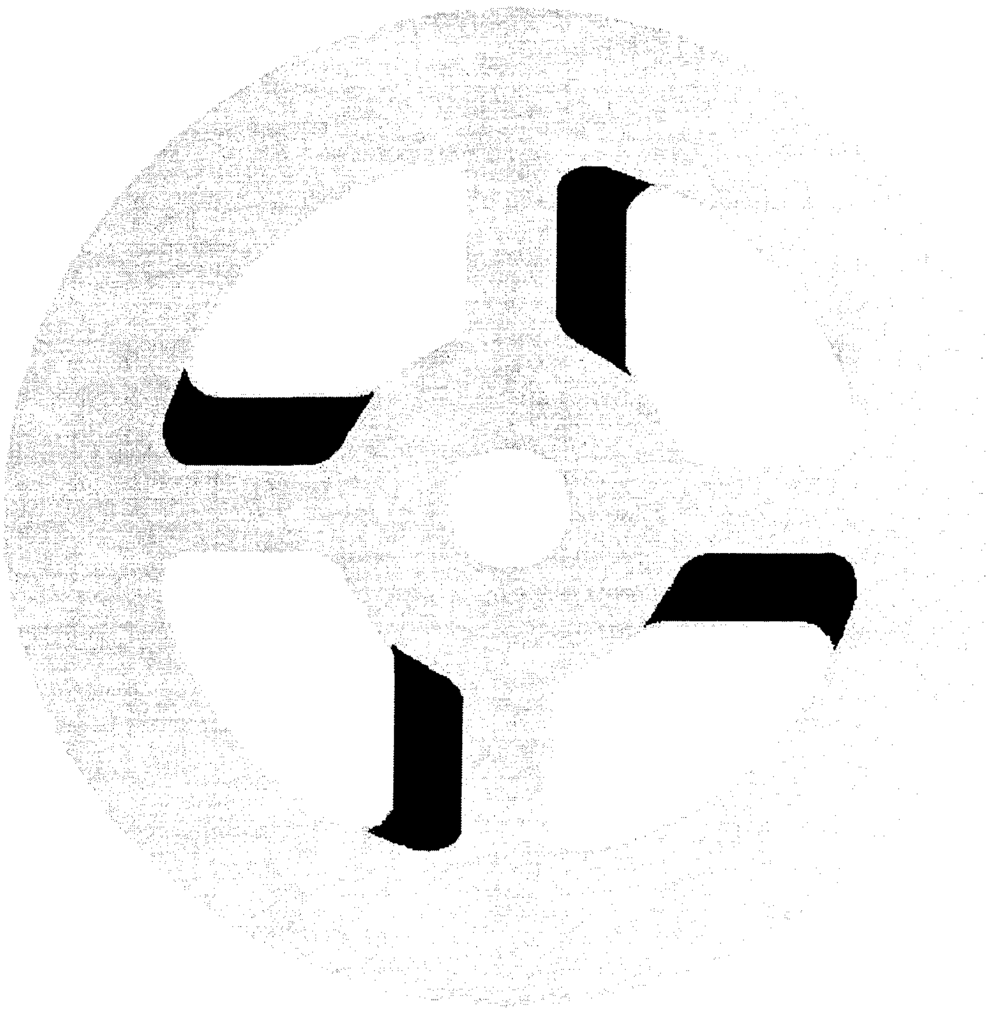


Figure 4.7. CAD model of the test workpiece that requires 5-axis machining (four symmetrically located inner contours require 5-axis cutting motion, for the other contours, 2-axis machining is sufficient).

- Design the CAD model of a waterjet cutting tool. According to the technical specification (Flow Automation Company), PASER3® waterjet cutting head has been modeled and attached to the robot's flange (end of joint five).

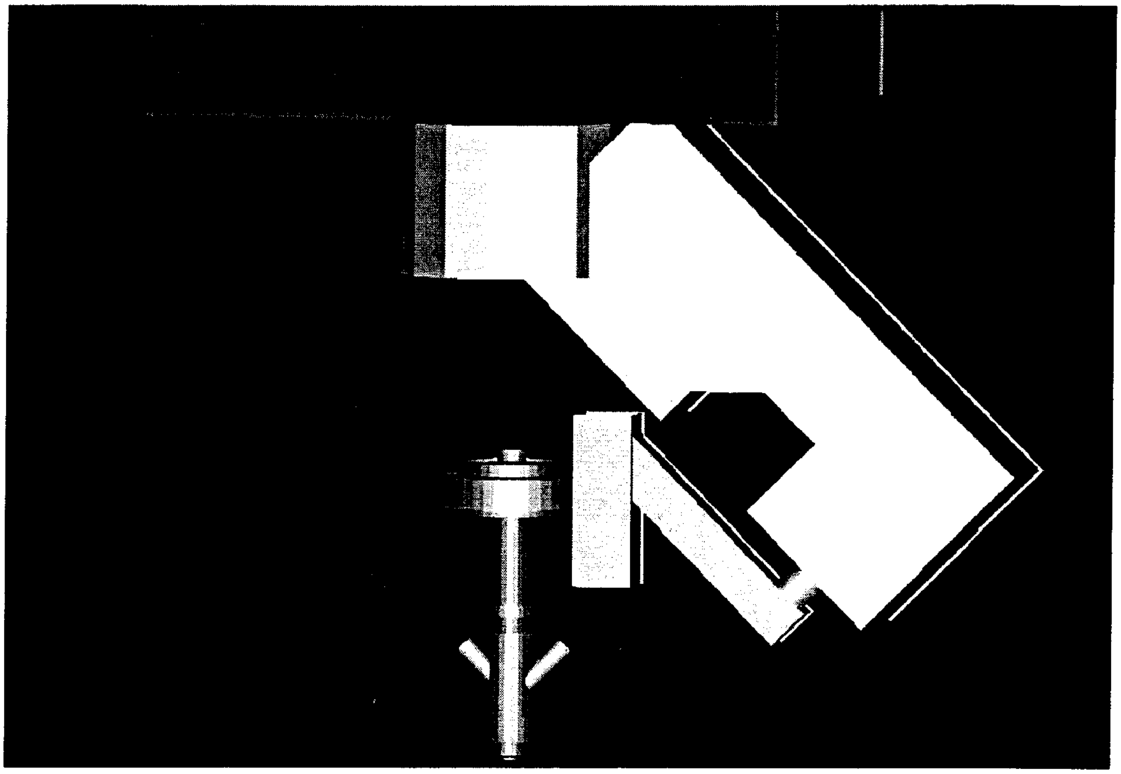


Figure 4.7. The CAD model of PASER3® waterjet cutting head attached to robot flange (end of Joint 5).

At the end of the presented procedure, the modeled robotic cell will contain all the information required by the simulation package:

- The Denavit-Hartenberg matrix defining the relationship between the robot joints.
- The name of the inverse kinematics template used by the robot.
- The total number of joints.
- The type of each joint (rotational or translational).
- Joint limits.
- Joint maximum velocities.
- Joint maximum accelerations.
- Robot zero position.
- Robot home position.
- CAD model of the workpiece that will be used for testing.
- CAD model of the waterjet cutting tool.

CHAPTER V

SIMULATION MODULE DEVELOPMENT

5.1. CREATON OF A ROBOT LANGUAGE TRANSLATOR

Language translators are software programs intended to take input files written in one language and to format the output files in different, destination language. Their area of application is wide, ranging from interpretation of natural languages to translation of computer programming languages into industry based machine specific programming languages.

In parallel with the improvements of the algorithms, methods, and techniques for defining and realization of translators, the appropriate software tools have been developed for automatic generation of their particular parts, the so-called compiler-compilers. The use of these programs intended for compiler generation facilitates the work on the translator construction, and frees the programmers from thinking about techniques and computer limitations with respect to translator implementation.

5.1.1. Language translators created by compiler-compilers

Compiler-compilers are used to generate language processors (such as compilers, translators, or interpreters) from high-level descriptions. After specifying the grammar of the language to be translated, the compiler-compiler creates a program that processes input text written in that language. This program hierarchically decomposes the input text into phrases. For each phrase the semantic actions can attached to grammar rules, which are elaborated when the corresponding phrase is processed. Semantic actions contain the code that matches the syntax of a language to be translated with the appropriate syntax of a destination language, and print out the destination language syntax in the output file.

Technology from classical compiler construction becomes more and more important in the fields of domain specific languages, document processing, and automatic software generation. Authors working in these areas are experts in their field, but not necessary experts in the field of parsing technology. Whereas most programmers are familiar with the notion of grammar scripts, it requires special education to write a grammar script in such a way that it fulfills the requirements of a specific parsing approach.

Compiler-compilers are based on interaction between a lexer and a parser. Section 2.4 describes the basic concepts of lexing and parsing. Also, Appendix A presents language syntax for parser generation, and Appendix B defines the most important parsing and lexing terms. Thorough understanding of sections mentioned is absolutely necessary in order to follow the work presented in this chapter.

The structure of a parser is composed of three sections, the definition section, the rule section, and the subroutine section.

The first section defines all the tokens expected to be received from the lexical analyzer. Also, this section is often used as a header section for local and global variables, functions and subroutines.

The rules section describes the actual grammar as a set of *production rules* or simply *rules.* Each rule consists of a single name on the left-hand side of the assignment operator ("="), a list of symbols and action code on the right-hand side, and a special symbol indicating the end of the rule. By default, the first rule is the highest-level rule. That is, the parser attempts to find a list of tokens, which match this initial rule, or more commonly, rules found from the initial rule. The expression on the right-hand side of the rule is a list of zero or more names. The symbol on the left-hand side of the rule can be used like a token in other rules. From this, the complex grammars can be built. Every rule is followed by the action code.

The action part of a rule consists of a blocks of code that can be written in one of the programming languages, depending on the compiler-compiler software utility used. Also, different compiler-compiler software packages execute action code at different times with respect to parsing phase. Thus, a parser can execute an action at the end of a rule immediately after that rule gets matched, or can completely execute the action code after the recognition of all the grammar rules.

The third and final section, is the user subroutine section. This section can contain any programming code written in appropriate programming language and is completely copied into the resulting parser. The minimal set of functions has to be provided in this section necessary for a parser and a lexer to compile: *main( )* and *error( ).* The *main* routine keeps calling the parser until it reaches the end of the input file. The *error* routine handles the input that cannot be parsed.

When a lexer and a parser are used together, the parser is the higher level routine. It calls the lexer whenever it needs a token from the input. The lexer then scans through the input recognizing tokens. As soon as it finds a token of interest to the parser, it returns to the parser, returning the token's value.

Not all tokens are of interest to the parser. In most programming languages the parser doesn't want to receive comments and whitespace for example. For these ignored tokens, the lexer doesn't return so that it can continue on to the next token without interacting with the parser.

5.1.2. Creation of G-code language translator

The purpose of the G-code language translation is to use the input file written in G-code language and automatically generate the matching code in Workspace Simulation Language. That custom-made simulation language then communicates with default motion planner and simulation engine in order to display the simulation of the G-code input file.

For the creation of the G-code language translator compiler-compiler software packages PRECCX and Flex were used. Therefore, the compilers were coded in C language, but not by hand. The source code for a C language compiler was automatically generated by Flex and PRECCX.

PRECCX, the "PREttier Compiler-Compiler" is very similar to much better-known compiler-compiler software YACC. Unlike YACC, PRECCX creates top-down parsers with infinite lookahead capability, parameters, arbitrarily complex compound expressions and synthesized attributes. Refer to Appendix A for more information about using PRECCX.

Flex is a popular tool for developing lexical analyzers. It takes an input robot program file and tokenizes it, that is, it identifies different strings as being keywords, numbers, identifiers, etc. These tokens are then passed on to the grammar parser, where they are used to match the semantic rules.

5.1.2.1. The translation process

Flex and PRECCX together form a compiler-compiler. Each takes an input file conforming to a special format, and outputs C source code. The output code of these programs is then compiled and linked to create an executable. That executable performs the translation from a native robot language into a Workspace Simulation Language module.

Figure 5.1 depicts the language translation process used. Language description represents the set of grammar rules that describe G-code language syntax. Compiler-compiler (designated with C-C symbol) software is a combination of Flex lexer and PRECCX parser generators. Source program is generated in ANSI C language. Compiler is the standard Visual C++ compiler for Windows® environment. Data identifier on the same figure is a general term for input file, which in this case is a G-code language part program. Program output is a translated file written in Workspace Simulation Language.

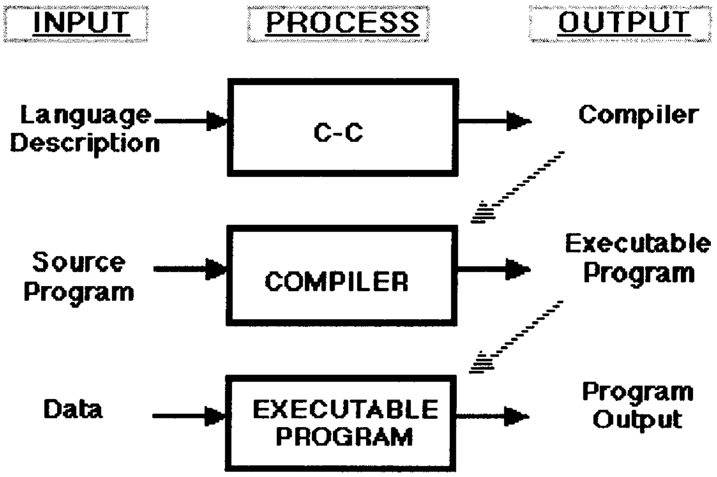


Figure 5.1. The flowchart that represents all the stages of language translation procedure, designating the inputs, processes and outputs.

The next three sub-sections will describe the major stages in language translator creation, focusing on the grammar coding and the solutions for the parser action execution and lexer ambiguities.

5.1.2.2. Writing the grammar rules to describe the syntax of G-code language

Describing the grammar of G-code language was not very difficult task from the programming perspective, but it required considerable amount of time to be completed.

Relative simplicity of the task was determined by moderately complex G-code language syntax. The language is composed of keywords mainly, with just a few variable data types. Also, the language functions and subroutines cannot accept any parameters, i.e. only void arguments can be passed, which makes the whole issue much simpler. However, the language is very extensive, with more than two hundred keywords and structures, each containing several arguments. The language abundance combined with relatively old-fashioned PRECCX debugger (the debugger works in DOS environment and its error locating accuracy is very low), made the whole task very time consuming and prone to errors. But, if the grammar was hand-coded in C programming language, the coding performance would have been definitely worse (more time, much more code, very prone to errors). G-code grammar written using PRECCX syntax rules is presented in Appendix E.

5.1.2.3. Solution for the PRECCX action execution problem

Parser actions represent blocks of code written in a programming language (in this case in C language) that are executed after particular grammar rule (or rules) has been matched. In this dissertation, the C action statements were primarily used to print into the output file Workspace Simulation Language function calls. With respect to that, section 5.2.1 was written to describe the main characteristics of simulation languages in general, while section 5.2.2 explains Workspace Simulation Language embedment into the action portion of the language translator.

This section presents how PRECCX parser generator was customized for G-code language translation. PRECCX must be customized because of its characteristic that the parsing is done first, and then the actions are performed. This means that all of the variables, which a developer has come to rely on in YACC to pass variable values to the action statements (*yytext*), may have been overwritten numerous times since the parse phase.

One may be thinking that he could just store a list of what tokens were read in which order, then when PRECCX worked through the actions, the next token in the list he could read off. The problem with this is that PRECCX goes backwards and forth through the parse checking out all possible grammar rules, before deciding on what it should do, so the actions may be repeated a large amount of times before doing anything useful. Additional set of C functions needed to be written in order to avoid incorrect output. The source code for those functions is presented in Appendix F.

The way around this is to make a list that always returns the right string. It can be done with the function:

```
Int ukey(int * String, int length )
```

All that needs to be done is to pass the token found at the beginning of a string and the length of a string. Then, the checking is performed to see if the string is already on the list. If so, a unique value is returned which can be, in turn, returned to PRECCX.

For instance,

```
@ integer = digit\x { digit *}\y   {@ $y ? ukey((int *) $x, $y – $x) : ukey((int *) $x, 1) @}
@ digit = (isdigit)      {@ (int) pstr @}
```

In the above code a value *pstr* is passed back with every digit. *pstr* is an integer pointer which PRECCX uses to know where it is in the parse. When an integer has been matched, the value returned by *{ digit *}* is checked (if there have been no occurrences, then it returns 0, otherwise it returns the value returned by the last occurrence). If the value in *$y* is non-zero, then the number's length is going to be the difference between the two pointers, otherwise the length will be 1.

The *ukey* function looks to see if the string is in the list, if so then it returns the position, if not, then it adds a new string to the list and returns its position. To use the string stored, the *char * getstring(int)* function is used, to which the *ukey* position of the string wanted to be read has been passed, and it returns the string value. If the value passed is zero or the value is beyond the end of the list *getstring* just returns an empty list.

So, that's how the terminals such as numbers, strings and comments are handled. Abstract syntax trees (AST) have to be done similarly, but not using the same functions. Instead of *ukey*, the *ASTukey* function is used, in much the same way, except it takes numerical values to point to the left and right branches. For example:

```
@ expr = number\x <'+'> expr\y {@ ASTukey($x, "+", $y) @}
@       | number\x      {@ $x @}

@ number = integer\x          {@ ASTukey(0, getstring($x), 0) @}
```

The simple *expr* expression always returns the output of an ASTukey. If the expression should be written into a file, then the *output_tree* function is used for that:

```
@ assignment = <Hello> <'='> expr\x {: fprintf(output_file, "\nHello = %s",
output_tree($x)); :}
```

Only the functions that return *int* can be put in *{@ @}* brackets, as these brackets themselves contain the return value used in a function. Semi-colons shouldn't be put after the statement, but there are mandatory after the statements contained within *{: :}*.

The PRECCX actions are attached to grammar rules and presented in Appendix E.

5.1.2.4. Solution for the Flex ambiguity problem

The disadvantage of the tradeoff made to gain PRECCX's infinite lookahead is that it lacks the typing capability of YACC. The result is that it is not possible to pass union structures in PRECCX. This creates some difficulty in the passing of tokens from the lexer to the parser, to the extent that it is not possible to do all lexical analysis with Flex. The recognition of identifiers and data literals must be done in PRECCX, leaving FLEX to recognize keywords and pass on other symbols one character at a time.

All of this means that numbers, string literals, comments and identifiers have to be handled through PRECCX instead of Flex, and this brings up a couple of problems.

The first one can be easily depicted by the following case: Translating the language that has a keyword called "*Print*" and an identifier called "*Printed*". If looking for identifiers using PRECCX, the Flex file will be something like:

```
Print          {nTemp = Print; output(nTemp); return nTemp;}
.              { nTemp = yytext; output(nTemp); return nTemp;}
```

The problem arises, when Flex inputs the following:

```
Printed = True
```

The tokens it passes to PRECCX will be:

```
<Print> <'e'> <'d'> <'='> ...
```

This will certainly cause problems, because the keyword "*Print*" will be recognized instead of indetifier "*Printed*". What needs to be done is to tell Flex to match the word *Print,* only if it's not followed by a character, which would make it an identifier. One way to deal with that is:

```
punct    [ \t\n"()<>,]
```

```
%%

Print/{ punct }                {nTemp = Print; output(nTemp); return nTemp;}
.                              { nTemp = yytext; output(nTemp); return nTemp;}
```

The meaning of "/" is that Flex will only match the rule, if it's followed by any symbol after the "/".

The second Flex problem is somewhat more complicated. Having the following identifier:

ImPrint

Flex would now match <*'I'*> <*'m'*> <*Print*>. The solution was found in using a feature called *exclusive start states*. That feature allows setting the rule in Flex that will always match a letter only as a letter, once it's started matching an identifier. This is done as follows:

```
%x     IDENT

%{
%}

punct   [ \t\n"()<>,]
%%


<IDENT> [A-Za-z0-9]|_)      { nTemp = yytext; output(nTemp); return nTemp; }
<IDENT>.                    { BEGIN INITIAL; unput(yytext[0]); /* allow this to be
                            checked by the initial state*/}
<IDENT> {eol}               { BEGIN INITIAL; unput(yytext[0]); /* allow this to be
                            checked by the initial state*/}
```

```
Print\{ pucnt }           {nTemp = Print; output(nTemp); return nTemp;}
[a-zA-Z]                  { BEGIN IDENT; nTemp = yytext[0]; output(nTemp);
                          return nTemp; }
.                         { nTemp = yytext; output(nTemp); return nTemp;}
```

The *%x IDENT* part declares an exclusive start state called *IDENT*, and the *<IDENT>* specifiers are the rules which Flex will only check when it has been placed in the IDENT start state.

When Flex is in its *INITIAL* start state it matches a single character, returns that character, and then places the program into the *IDENT* start state. The *IDENT* start state then works through all of the stream, passing back the single characters, until it reaches a character which is not a valid character for an identifier. When it finds such a character it places it back on the stack (to allow it to be matched by the *INITIAL* start state) and then switches back to the *INITIAL* state before carrying on again.

5.2. ADDING SIMULATION LANGUAGE FUNCTION CALLS TO ACTION STATEMENTS OF THE LANGUAGE TRANSLATOR

A simulation language represents an interface between a programmer, at one side, and a motion planner and simulation engine of a simulation software package on the other side. A program written in a simulation language is interpreted by simulation software and significant parameters are then sent to the software's motion planner to calculate the motion trajectory and motion parameters, as well as to the simulation engine to generate a list of simulation events. Simulation languages can be structured or object-oriented.

5.2.1. Simulation languages in general

Structured simulation languages offer prespecified functionality produced in another language (assembly language, C, FORTRAN, etc.) and the user cannot access the internal mechanisms within the language. Instead, only the vendor can make modifications to the internal functionality. Reusing language features requires that the user code any new features as though they were a completely separate package. Therefore, full integration with the existing language is not possible.

Also, users have only limited opportunity to extend an existing language features. Some simulation languages allow for certain programming-like expressions or statements, which are inherently limited. Most languages allow the insertion of procedural routines written in other general-purpose programming languages. However, none of these procedures can, in any way, become an inherent part of the preexisting language.

Thus none of these approaches is fully satisfactory because, at best, any procedure written cannot use and change the behavior of a preexisting object class. Also, any new object classes defined by a user in a general programming language do not coexist directly with vendor code.

Object-oriented simulation deals directly with the limitation of extensibility by permitting full data abstraction as well as procedural abstraction. Data abstraction means that new data types with their own behavior can be added arbitrarily to the programming language (abstract data types). When a new data type is added, it can assume a role as important as any implicit data types. For example, a user-defined data type that manages complex numbers can be as fundamental to a user's language as the implicitly defined integer data type. In the simulation language context, a new user-defined robot class can be added to a language that contains standard resources without compromising any aspect of the existing simulation (Banks et al., *Discrete Event System Simulation*, 1996).

The advent of Visual Basic, C++, Java, and other object oriented programming languages facilitate a fundamentally different approach to the design and implementation of simulation software that specifically addresses the shortcomings of existing tools. In particular, the combination of process-oriented simulation, an object-oriented programming language, and the Component Object Model (COM) software architecture will allow models to be packaged in a way that increases accessibility to the user without the compromises required by increased separation of the user from the underlying modeling language.

Despite common origins and the ideal suitability of object-oriented methods to the task of structuring process models, the simulation community has been slow to adopt object-oriented methods. A contributing factor has been a lack of commercial simulation software tools with coherent and accessible support for object-oriented process modeling. In fact, an appreciation for object-oriented methods and their attendant benefits requires only the understanding of a few simple concepts; namely, *encapsulation, classes, messages,* and *inheritance.*

Objects and their software implementation are patterned after real-world objects. They have data (attributes, characteristics, properties, etc.) that represent the state of the object and a set of behaviors that describe the ways in which they can be operated on. In an object-oriented approach, the association between the state of an object and its set of behaviors is made explicit via encapsulation whereby both are defined in an integral, self-contained unit called a class.

This collective definition serves as a template or blueprint for creating particular instances of the corresponding class. Each instance (of which there may be many) possesses its own unique copy of the state-related data defined by the class but shares the behaviors. Communication between objects is confined to a formalized system of messages. It is convenient to think of message generation and processing as just additional types of behaviors defined on the sending and receiving classes. Finally, inheritance is a mechanism by which new classes can be defined as extensions of existing ones. The derived class has all the characteristics and behaviors of the parent class (which may itself be derived from others) plus some added functionality in the form of new characteristics and/or behaviors.

It is easy to see in these concepts two particularly natural applications to process-oriented simulation modeling. One is the use of encapsulation to make explicit the association between the representation of an entity and its sequence of processing steps. Another is the encapsulation of extended sequences of low-level processing steps into sub-models whose behavior can be invoked as a single high-level processing step by instances of a desired entity class. These two notions are central to the design of Workspace Simulation Language (WSL) which implements an object-oriented, process-modeling capability within the framework of the Visual Basic for Applications (VBA) programming language.

5.2.2. The output from the language translator – Workspace Simulation Language (WSL)

At the internal level of Workspace, all robots are considered equal. That is, every robot model is controlled with the same simulation language, and every robot can perform the same operations. This contrasts sharply to real life, where each brand of robot uses a proprietary language, and may perform different operations than other manufacturer's robots.

An end user of Workspace should be able to load a program written in a robot's native language and see the Workspace model perform just as a real robot would, if executing that program. It is here that translation becomes necessary: before the robot's program can be simulated, it must be converted from the proprietary native language into the simulation language used by Workspace.

Workspace Simulation Language is the custom made simulation language built upon the foundations of Visual Basic for Applications language. In other words, highly customizable VBA language has been complemented with several robotics related classes.

The language translator, described in section 5.1, in its action statements contains the source code written in C programming language that prints out appropriate Workspace Simulation Language instructions (for each of the matching robot language instructions) into the output file. That output file is then used to control the overall behavior of the robot model in simulation.

Robotics related classes in Workspace Simulation Language were implemented according to the recommendations of Realistic Robot Simulation (Realistic Robot Simulation) specification, therefore the output statements from the language translator contain programming calls to Realistic Robot Simulation-based class members (variables and methods).

The Workspace Simulation Language functions are defined in the motion planner and therefore coded in C++ programming language. The communication between the Workspace Simulation Language and the motion planner is established through the Component Object Model interface.

Figure 5.2 depicts the robot language translation process, followed by the simulation of the translated program, showing the parts designed and implemented by the author.

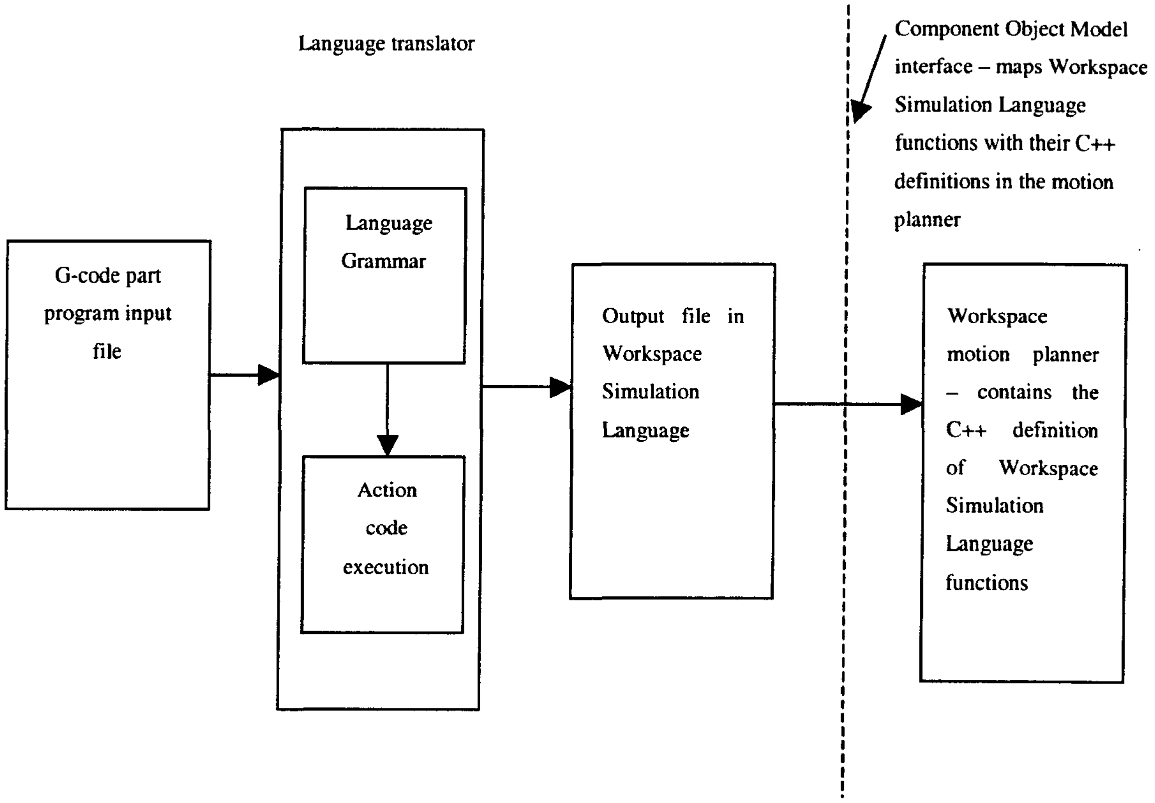

Language translator
Component Object Model interface – maps Workspace Simulation Language functions with their C++ definitions in the motion planner
G-code part program input file
Language Grammar
Action code execution
Output file in Workspace Simulation Language
Workspace motion planner – contains the C++ definition of Workspace Simulation Language functions

Figure 5.2. The relationship between the language translator and the Workspace motion planner established through the Component Object Model interface.

In the next two subsections (5.2.2.1 and 5.2.2.2) robotics-related instructions of Workspace Simulation Language will be described (based on the recommendations of Realistic Robot Simulation specification), as well as the Visual Basic for Applications language function library written to facilitate the embedment of Workspace Simulation Language into the action portion of the language translator.

5.2.2.1. Realistic Robot Simulation services as a part of Workspace Simulation Language

Realistic Robot Simulation interface specifies almost two hundred services, more than any one robot can actually use. Robot manufacturers who support Realistic Robot Simulation provide a subset of the services, based on the capability of their robots. Examples of services provided by Realistic Robot Simulation are the setting of a speed and acceleration, kinematics operations, (conversion between Cartesian and robot joint values) and motion execution.

The main Realistic Robot Simulation services used for robot language translation are:

- *GetInverseKinematics* – used to compute inverse kinematics.
- *GetForwardKinematics*- used to compute forward kinematics.
- *GetCellFrame*- used to give information about specified cell frame.
- *ModifyCellFrame* - used to modify robot's Tool and Base frames.
- *SelectWorkFrames*- used to select from the predefined frames (object and tool frames).
- *SelectTargetType*- used to select one of the different types for the specification of targets.
- *SetMotionType* – sets the motion type to be Linear, Circular or Joint.
- *SetJointSpeeds*- for each specified joint sets the joint speed.
- *SetJointAccelerations* - used to set the accelerations of the individual joints.
- *SetCartesianPositionSpeed*- used to set the speed for the Cartesian motion.
- *SetCartesianOrientationSpeed*- used to set the orientation speed during Cartesian motion.
- *SetCartesianPositionAcceleration*- used to set the acceleration for the Cartesian motion.
- *SetCartesianOrientationAcceleration*- used to set the orientation acceleration during

Cartesian motion.

- *SelectFlybyMode-* used to set flyby mode on or off.
- *SelectPointAccuracy* – selects the motion tolerance level.
- *StopMotion-* used to stop the on-going motion toward the target.
- *SetMotionTime-* used to specify the motion time to the next target, instead of specifying the motion speed.
- *SetRestParameter-* used to set the dwell time between motions.
- *SetNextNamedTarget* – used to specify the next motion target.

The following Realistic Robot Simulation services are performed automatically in Workspace:

- *Initialize,*
- *Terminate*, and
- *GetNextStep.*

Initialize and *Terminate* services are called automatically when the robot object is created or destroyed, while *GetNextStep* is called by the simulation engine after calling *SetNextNamedTartget* in order to perform the motion of the robot.

Due to the mismatch of data types of Realistic Robot Simulation specification and of Workspace Simulation Language, which is based on Visual Basic for Applications language, the following conversion table was used:

RRS Specification	VBA in Workspace
bitstring	Integer
int	Integer
string	String
real	Double
CartposType	RCSCartPos
JoinPosType	RCSJointPos

Table 5.1. Data type conversion table between Realistic Robot Simulation interface and Visual Basic for Applications language data types.

RCSJointPos and *RCSCartPos* are both objects and as such should be defined in Workspace Simulation Language by using *New* keyword.

5.2.2.2. Creation of the Visual Basic for Applications language library of supporting functions

Ultimately, the Workspace Simulation Language code output by the compiler must call the Workspace motion planner through the Component Object Model interface to perform the robot motion. Section 5.3.1 gives an insight in robot motion planning in general, whilst section 5.3.2 explains the principles of motion planning in Workspace and shows how the most important motion planning functions were created.

The alternative and obviously shorter way to connect the language translator with the motion planner is that the language translator calls functions of the Workspace motion planner directly, thus eliminating the need for the Component Object Model interface involvement. However, this solution requires a fair amount of C code to be added to the parser. Much of this code would be numeric/string conversions, and would require many global variables to be introduced to the parser. Such code would almost certainly be tedious to write, and it would detract from the readability of the parser code.

Also, if the language translator were to output the Workspace Simulation Language statements only, the developer would, again, need a great deal of C code embedded in the parser.

Here is the related example for the robot linear motion command:

RMoveLinear RPosition

RPosition structure uses *x, y, z* coordinates and *roll, pitch, yaw* angles to store a robot position and orientation. Realistic Robot Simulation specification, however, defines Cartesian positions as parameters of a Denovitt-Hartenberg's matrix stored in Realistic Robot Simulation *RRSCartPos* object. Therefore, *RMoveLinear* must covert the 6-variable *RPosition* into a 12-variable *RRSCartPos*, and then use that variable as the argument to *RRS.SetNextTarget* to perform the robot motion. To convert the values from one representation into the other, the following actions should be undertaken: declaration of the global variables for the *RPosition*, conversion of tokens into numbers, assignment of those numbers to the global variables, conversion of the position types, conversion of the resulting numbers back into strings, and output of the strings in form of Workspace Simulation Language function calls. In this method, the tasks are not split across subroutines. The resultant parser code will suffer symptoms of code bloat: it is less readable, less maintainable, and more prone to logic errors.

An easier, more modular way to code is to create a library of wrapping functions in Visual Basic for Applications language. Using Visual Basic for Applications language, these tasks are easily split across several subroutines, part of a library that can be easily read and maintained.

Workspace Simulation Language function calls, then, should always be wrapped in Visual Basic for Applications language subroutines with syntax as close to the original native robot language as possible. The products of this procedure are parser code that is more readable to developers, and translated Workspace Simulation Language code that is more readable to end-users.

The library of the Visual Basic for Applications language wrapping functions has been developed by the author of this dissertation and presented in Appendix G.

The next section (5.3) defines motion planning in general and presents the part of the Workspace motion planner developed by the author.

5.3. IMPLEMENTATION OF THE WORKSPACE SIMULATION LANGUAGE FUNCTIONS IN WORKSPACE MOTION PLANNER

5.3.1. Robot Motion Planning in general

In any robot task the robot has to move from a given initial configuration to a desired final configuration. Except for some special cases, there are infinitely many motions for performing the task. Even in complex tasks, where the interactions of the robot with the environment may impose additional constraints on the motion, the set of all possible motions is still very large. Motion planning is the process of selecting a motion from the set of all possible motions while ensuring that all constraints are satisfied.

Motion planning can be viewed as a set of computations that provide sub-goals or set points for robot control. The computations and the resulting motion plans are based on a suitable model of the robot and its environment. The task of getting the robot to follow the planned motion is called *control.*

The motion of a robot system can be described in two different spaces. First, the task is specified in the so-called task space or Cartesian space. It is customary to use Cartesian X, Y, and Z coordinates to specify the position of a reference point on the end-effector of a robot with respect to some absolute coordinate system and some form of Euler angles to specify the orientation. However, for a multi-degree-of-freedom robot, specifying the position of the end-effector may not specify the position of all the robot links. For an n-degree-of-freedom robot, it may be necessary to specify the robot motion in the joint space by specifying the motion of n independent robot joints. The joint space is the Cartesian product of the intervals describing the allowable range of motion for each degree of freedom.

Based on this classification, it is easy to see that it is possible to define a motion planning problem in the task space or in the joint space. In order for a robot to perform the task of, say, welding, it may be sufficient to plan the end-effector trajectory. That requires a motion plan in the task space. Such a motion plan may be satisfactory if there are no obstacles in the environment and the dynamics of the robot do not play an important role. However, if there are obstacles or if the robot arm has more than six degrees of freedom, it may be necessary to plan the motion of the arm in the joint space. Such a motion plan would guide the robot around obstacles while guaranteeing the desired end-effector trajectory for the welding task. It is easy to see that motion planning in the joint space can take constraints in the task space into account, but not the other way around. In other words, given a motion plan in the task space, it is possible to solve for a compatible motion plan in the joint space that accommodates additional constraints, particularly for a kinematically redundant robot, where the number of robot's degrees of freedom is greater than six ($n > 6$).

5.3.2. Motion Planning in Workspace

Motion planning focuses on the controlling a manipulator motion so that it follows a preplanned path (A path is a curve in three dimensional space that the manipulator hand moves along from the initial position and orientation to the final location).

Before moving a robot arm, it is of considerable interest to know whether there are any obstacles present (obstacle constraints) and whether the manipulator must follow a specified path (path constraints). These two constraints give rise to four possible control modes as described in the following table:

		Obstacle Constraint	
		Yes	No
Path Constraint	Yes	Collision-free path planning plus path tracking	Path planning plus path tracking
	No	Positional control plus obstacle detection and avoidance	Positional control

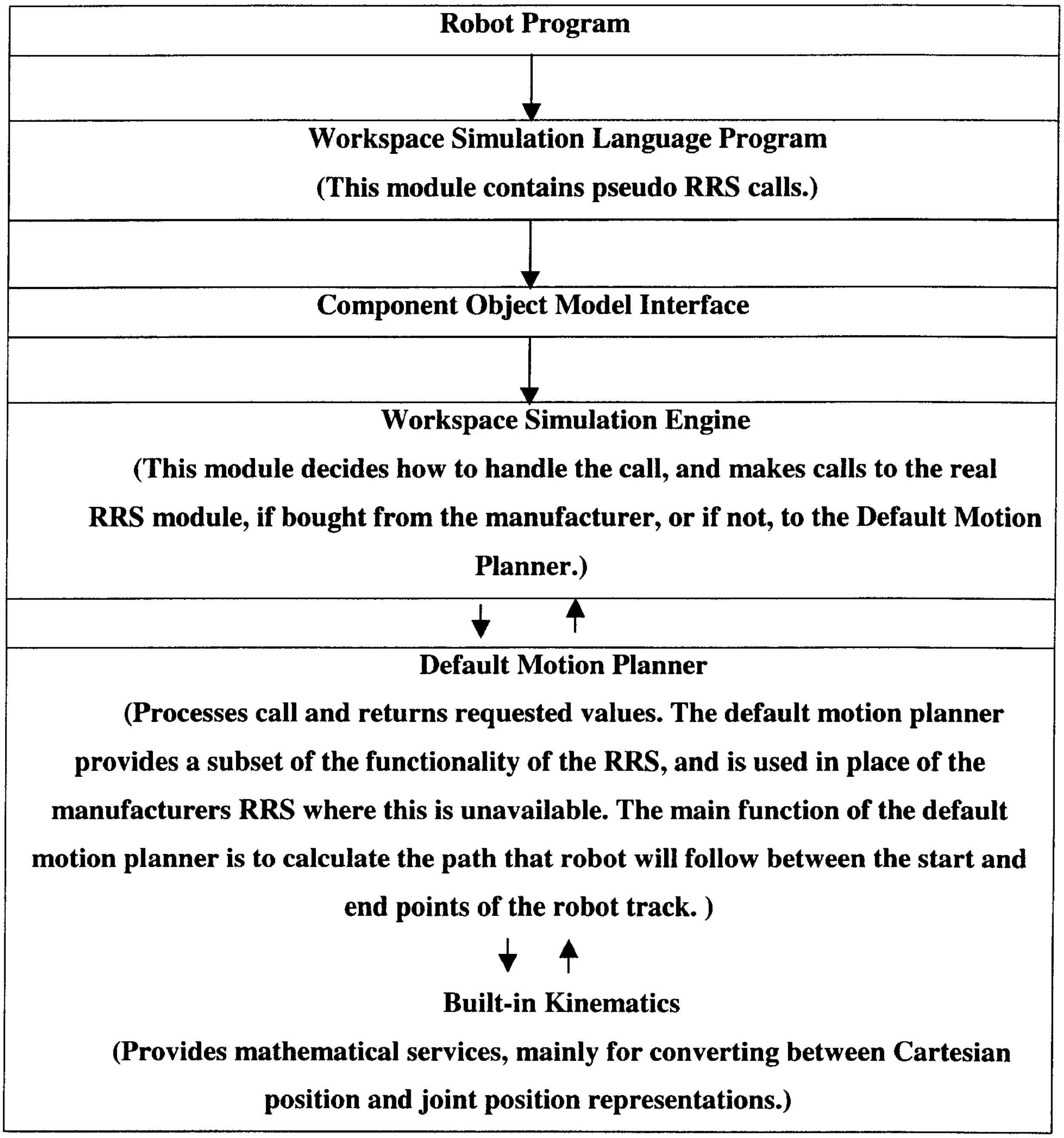
Table 5.2. Four possible control strategies in motion planning depending on presence of path and obstacle constraints.

Since one of the main features of Workspace is collision detection, then the obstacles are not considered in determining the motion of a robot arm. It should be noted that a large portion of robotics research currently focuses on the planning of paths with obstacle constraints.

In trajectory planning the input is in form of variables (such as: motion type, fly by, point accuracy, target position, the manipulator dynamic constraints, etc.), which specify the path constraints and the output represents a sequence of time-based intermediate manipulator configurations (expressed either in joint or Cartesian coordinates).

The Workspace motion planner discussed here is also called "Default Motion Planner". The term "default" refers to the fact that Workspace relies on the user to have an Realistic Robot Simulation module from a specific robot manufacturer. Thus, if no Realistic Robot Simulation module is present, either if the user chose not to purchase one, or none was available from the manufacturer, then the default motion planner is used.

The default motion planner interacts with other building blocks of Workspace in the following way (Figure 5.3):



Robot Program

Workspace Simulation Language Program

(This module contains pseudo RRS calls.)

Component Object Model Interface

Workspace Simulation Engine

(This module decides how to handle the call, and makes calls to the real RRS module, if bought from the manufacturer, or if not, to the Default Motion Planner.)

Default Motion Planner

(Processes call and returns requested values. The default motion planner provides a subset of the functionality of the RRS, and is used in place of the manufacturers RRS where this is unavailable. The main function of the default motion planner is to calculate the path that robot will follow between the start and end points of the robot track.)

Built-in Kinematics

(Provides mathematical services, mainly for converting between Cartesian position and joint position representations.)

Figure 5.3. The default motion planner interactions with other software modules of Workspace

The simulation engine calls the motion planner. This is done through so called Realistic Robot Simulation shell. The Realistic Robot Simulation shell is the parent abstract class from which various objects are derived. The reason to create the various objects is to call the same function in various Realistic Robot Simulation modules built by robot manufacturers or in a default motion planner, if the Realistic Robot Simulation module is not available.

Overview of the general default motion planning algorithm is presented below in a step-by step form as well as in a graphical form on Figure 5.4:

- A Realistic Robot Simulation shell (default shell) is created. It is a member of the Robot class.
- Then in the shell's *Initialize* method a new *CDefaultMotionPlanner* object is made. The initial position of the robot with the *Set_Initial_Position* function is specified.
- When one wishes to specify a target, then he sets up a Cartesian and Joint position, and passes them onto *Set_Next_Target*. Only one, Cartesian or Joint position is to be valid.
- Then during a simulation the *Get_Next_Step* function is called which determines where the robot is at a specific time. During the first simulation interval the time required to perform the move is calculated. Having calculated the travel distance and taking into account the robot's acceleration and velocity, the travel time is determined. Then with any following call to *Get_Next_Step* the current time is compared with total travel time, and then the required position to move the robot to is determined.
- The motion type to be performed when traveling to a target is selected with the *Select_Motion_Type* service. The following types of motion are performed:

1. **Linear**: The trajectory of the Tool Center Point (TCP) follows a straight line from its current position to the required target position.
2. **Joint**: The joint values of both current and target positions have to be calculated (by using the inverse kinematics) and then interpolated, such that all the joints start and finish their motion at the same time.
3. **Circular**: To specify a circular trajectory three parameters need to be chosen from the following group: radius, centre point, start and end tangent, start, end and intermediate point on a circle. The three point circle representation (the start position, the target position, and a position, which is located somewhere on the circle and is between these two locations) has been chosen to be the default representation, and

from that, all the other parameter combinations can be derived. Intermediate point is usually referred to as a "via point".

The default motion planner source code is presented in Appendix H.

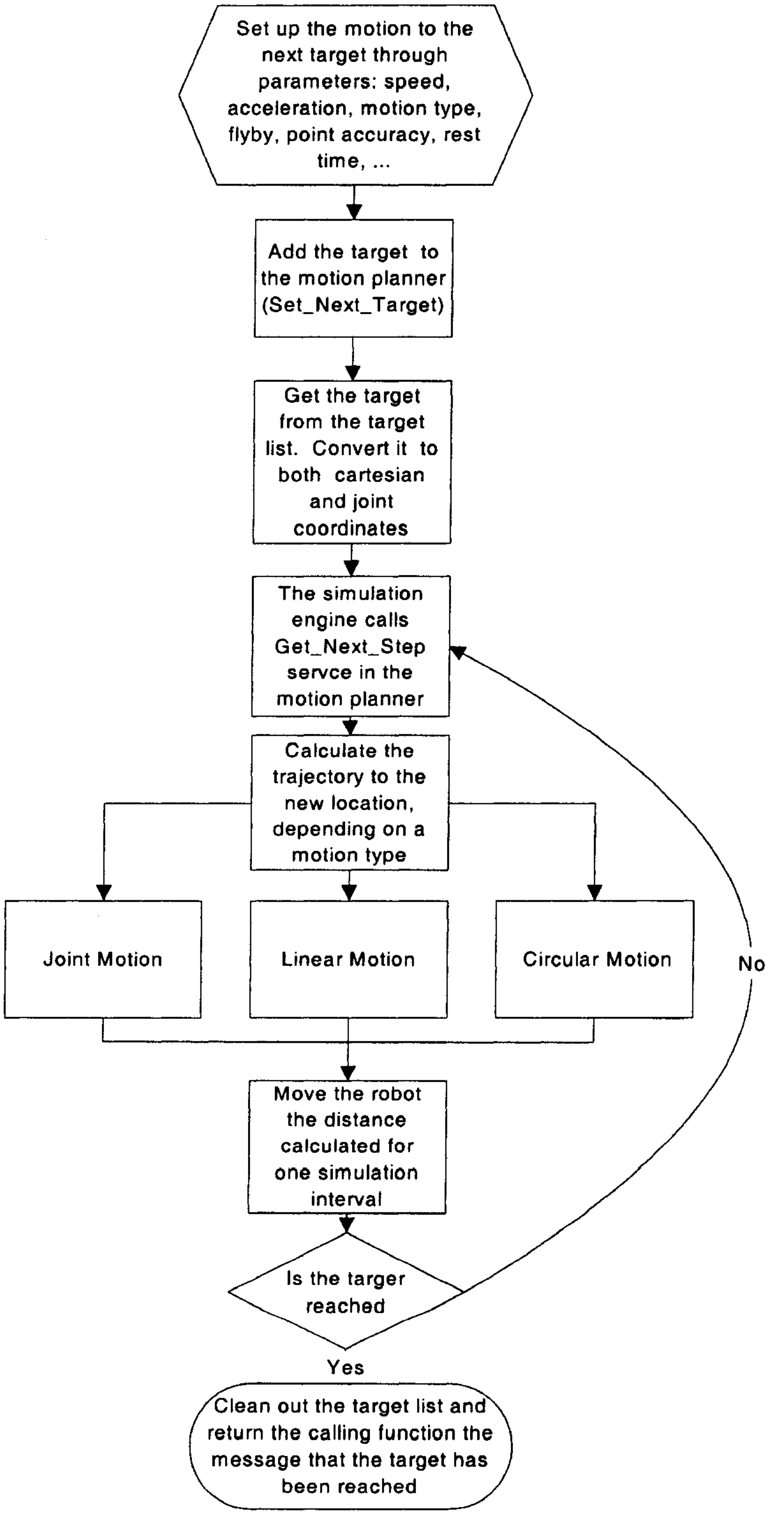

Set up the motion to the next target through parameters: speed, acceleration, motion type, flyby, point accuracy, rest time, ...
Add the target to the motion planner (Set_Next_Target)
Get the target from the target list. Convert it to both cartesian and joint coordinates
The simulation engine calls Get_Next_Step servce in the motion planner
Calculate the trajectory to the new location, depending on a motion type
Joint Motion
Linear Motion
Circular Motion
No
Move the robot the distance calculated for one simulation interval
Is the targer reached
Yes
Clean out the target list and return the calling function the message that the target has been reached

Figure 5.4. The general default motion-planning algorithm developed by the author.

- The velocity and acceleration of the robot are set through the functions:
  - *Set_Cartesian_Position_Speed,*
  - *Set_Cartesian_Orientation_Speed,*
  - *Set_Joint_Speeds, Set_Joint_Accelerations,*
  - *Set_Cartesian_Position_Acceleration, and*
  - *Set_Cartesian_Orientation_Acceleration.*
- Fly-by and point accuracy services are set by the following Realistic Robot Simulation services:
  - *Select_Flyby_Mode,*
  - *Set_Flyby_Criteria_Parameter,*
  - *Select_Flyby_Criteria,*
  - *Cancel_Flyby_Criteria,*
  - *Select_Point_Accuracy, and*
  - *Set_Point_Accuracy_Parameter.*

5.3.2.1. Trajectory velocities and accelerations

When in motion, a manipulator accelerates until it reaches programmed speed, then proceeds its motion at this speed, and finally decelerates until the speed of its tool center point (TCP) becomes zero. Both, robot and CNC controllers, define acceleration and deceleration phases to be equally long. Therefore, if the total distance between start and target points is greater than twice the distance required to reach the programmed velocity, the velocity profile will be of the following form (Figure 5.5.)

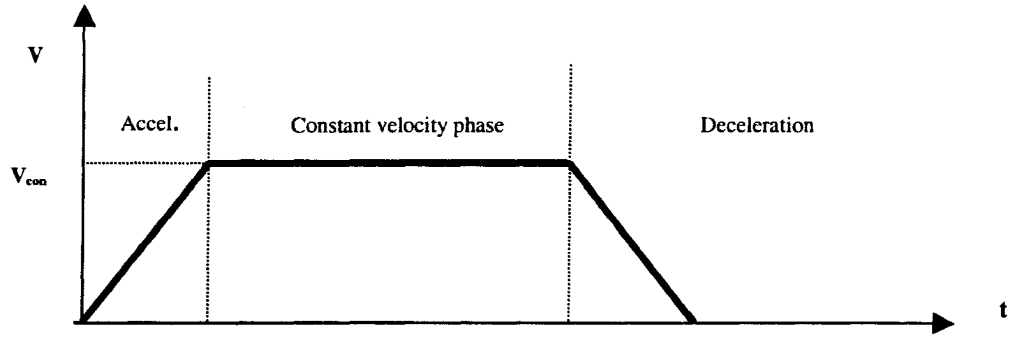


Figure 5.5. A manipulator's velocity profile when the total distance between start and target points is greater than twice the distance required to reach the programmed velocity.

If the total distance between start and target points is less then or equal to twice the distance required to reach the programmed velocity, the velocity profile will be of the form shown in Figure 5.6. That usually happens when manipulator's axial displacement is small and acceleration time significant, therefore constant velocity phase might not be achieved, so the speed profile would be composed of acceleration and deceleration phases only.

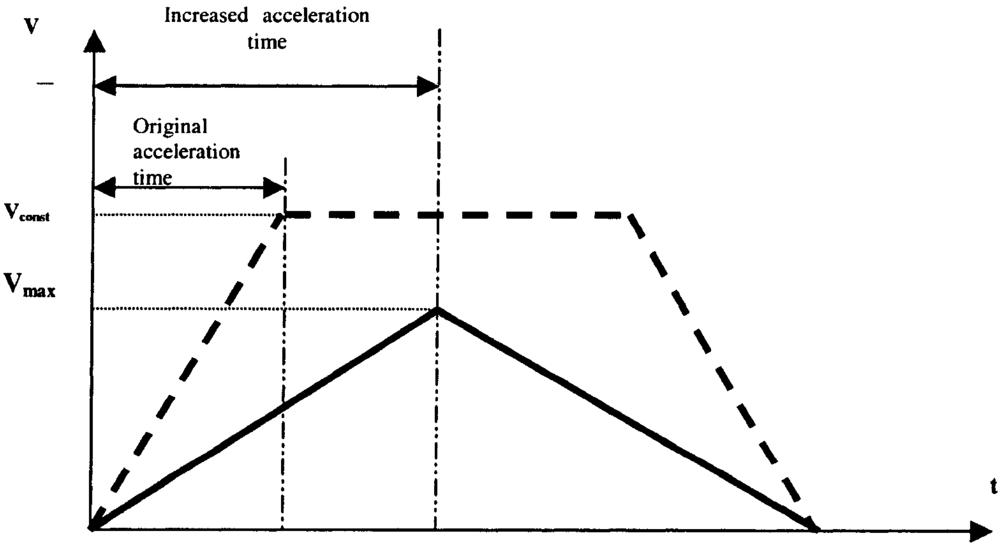


Figure 5.5. A manipulator's velocity profile when the total distance between start and target points is less then or equal to twice the distance required to reach the programmed velocity.

5.3.2.2. Trajectory calculation

Workspace supports three types of motion: joint, linear, and circular. The type of motion determines the tool center point from the current to the next specified target (and for any target thereafter until the motion type is changed) is set by the Realistic Robot Simulation function *Select_Motion_Type*.

Joint Motion

Joint motion (frequently called point-to-point or PTP for short) is the quickest way of moving the tip of the tool (Tool Center Point or TCP for short) from the current position to programmed destination position. To do this, the controller calculates the necessary angle differences for each axis. Joint motion simultaneously diminishes mechanical stress on the robot since the motor and the gear torque are reduced for all axes with shorter trajectories.

The movements of the axes are synchronized in such a way that all of the axes start and finish moving at the same time. This means that only the axis with the longest trajectory, the so-called leading axis, is actually moved with the programmed limit value for acceleration and velocity. All other axes move only with the velocity and acceleration rates necessary for them to reach the end point of the motion at the same moment.

The user can still specify the start and target positions in Cartesian coordinates, but they are converted into joint-variable space (using the inverse kinematics) when determining the path. Upon the first interpolation interval the amount that each axis needs to move and the amount of time it will take to perform its motion gets determined. In determining this: the distance, joint accelerations and joint velocities specified by: *Set_Joint_Accelerations* and *Set_Joint_Speeds* are used (Figure 5.6).

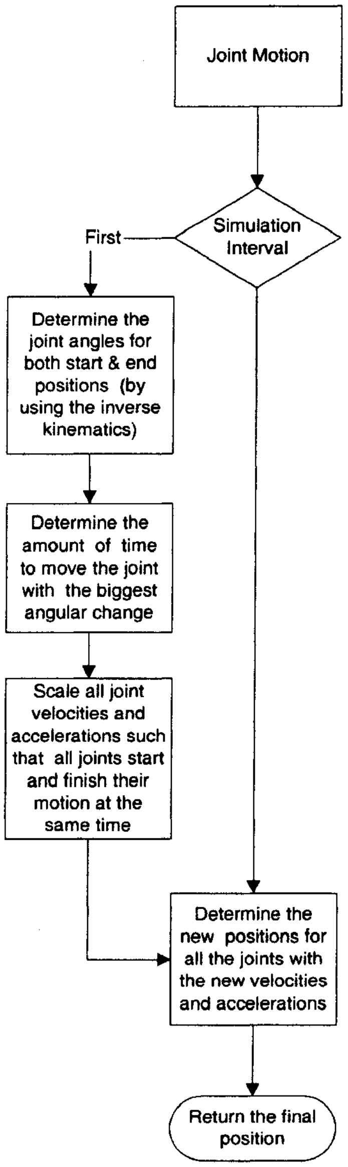

Joint Motion
Simulation Interval
First
Determine the joint angles for both start & end positions (by using the inverse kinematics)
Determine the amount of time to move the joint with the biggest angular change
Scale all joint velocities and accelerations such that all joints start and finish their motion at the same time
Determine the new positions for all the joints with the new velocities and accelerations
Return the final position

Figure 5.6. Joint interpolation algorithm (the first block, designated by "Joint Motion" is a part of the general default motion planning algorithm shown on Figure 5.4).

Linear Motion

Linear motion is the process of moving along a straight line that connects the start and target positions. Again the user can set the points in either Cartesian or joint coordinates, however the actual trajectory is determined in Cartesian coordinates. Upon the first simulation interval the distance between the teach points is calculated and if the motion speed is set, the time needed to perform the motion will be determined (Figure 5.7). This is done by using the distance between the two points in Cartesian space equation, and by using the speeds and accelerations set by the Realistic Robot Simulation functions:

- *Set_Cartesian_Position_Speed,*
- *Set_Cartesian_Orientation_Speed,*
- *Set_Cartesian_Position_Acceleration*, and
- *Set_Cartesian_Orientation_Acceleration*, to find the motion time.

The trajectory is then, determined by:

- Finding a vector between the start and end points.
- Normalizing this vector (this determines the unit vector equation).
- Scaling this vector by the value of calculated distance that robot's tool centre point needs to be moved for each simulation interval during: acceleration, deceleration and constant velocity phases.

The equations used for the procedure explained above were:

- Linear distance between two points (*New* and *Current*) in Cartesian space

$$LinearDistance = \sqrt{(New.x - Current.x)^2 + (New.y - Current.y)^2 + (New.z - Current.z)^2}$$

- Vector between the start and end point of the motion

$$Vector.x = New.x - Current.x$$
$$Vector.y = New.y - Current.y$$
$$Vector.z = New.z - Current.z$$

- Normalized vector (unit vector)

$$Norm = \sqrt{Vector.x^2 + Vector.y^2 + Vector.z^2}$$
$$UnitVector.x = Vector.x / Norm$$
$$UnitVector.y = Vector.y / Norm$$
$$UnitVector.z = Vector.z / Norm$$

- Acceleration time

$AccelerationTime = ProgrammedVelocity/ProgrammedAcceleration$

- Distance to reach programmed velocity

$$DistanceToAccelerate = \frac{ProgrammedVelocity^2}{2 \cdot ProgrammedAcceleration}$$

- Positional displacement for *n*-th simulation interval during the acceleration phase

$$\mathrm{D}_{\mathrm{AC}_\mathrm{n}} = \frac{ProgrammedAcceleration \cdot (n \cdot SimulationIntervalDuration)^2}{2} - \mathrm{D}_{\mathrm{AC}_{\mathrm{n-1}}}$$

- Simulation interval number

$n = 1 \div n_{\max}$

$$n_{\max} = \frac{AccelerationTime}{SimulationIntervalDuration}$$

- Positional displacement for each simulation interval during the constant velocity phase

$$D_{CON} = \frac{1}{n_{\max}}\left(LinearDistance - 2 \cdot DistanceToAccelerate\right)$$

- Positional displacement for *n*-th simulation interval during the deceleration phase

$D_{DCn} = D_{AC(n_{\max}-n)}$

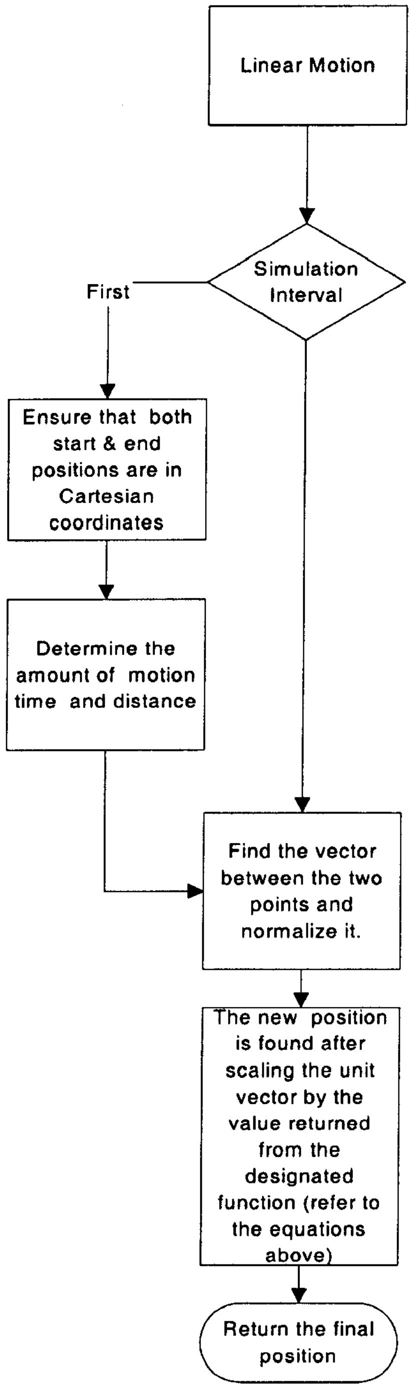

Linear Motion
Simulation Interval
First
Ensure that both start & end positions are in Cartesian coordinates
Determine the amount of motion time and distance
Find the vector between the two points and normalize it.
The new position is found after scaling the unit vector by the value returned from the designated function (refer to the equations above)
Return the final position

Figure 5.7. Linear interpolation algorithm (the first block, designated by "Linear Motion" is a part of the general default motion planning algorithm shown on Figure 5.4).

Circular Motion

Circular motion is quite similar to linear motion but instead of being bounded to a straight-line path, an arc-shaped trajectory is calculated. Also the user can set the points in either Cartesian or joint coordinates, however the actual trajectory is determined in Cartesian coordinates. What distinguishes circular motion from other motion types is that an additional parameter is needed, beside the target point, to determine the trajectory. If no additional parameter such as radius, arc centre, tangent or another point on the arc were provided, then an infinite amount of arc-shaped paths would be possible to be constructed through two points. In the default motion planner, a third point on the arc, located somewhere in between the start and the target point (called the via point), is used to uniquely define the circular trajectory.

In order to find the position and orientation of the tool centre point during each simulation interval of circular motion, the three points on a circle need to be relocated (by one translation and three rotations) so that the start position overlaps with the origin, the via point is placed on the X axis, and the arc lays in the XY plane. Two-dimensional trigonometry is then used to find the centre and radius of the circle. At each interpolation interval the position reached on the arc is then transformed back to the original coordinate system by a reverse transformation process.

Once the target and via data have been specified, the arc centre, radius and central angle to interpolate over are determined. The total time needed to perform the motion along the arc distance (arc central angle multiplied by the arc radius) with the specified tool center point velocity and acceleration is then determined. Then for every simulation interval the portion of circular arc that should have been traversed for the specified time interval is calculated.

Circular motion gets differently interpreted in CNC and robot controllers. Because of that, particular point of difficulty lays in mapping one interpolation representation to the other. Refer to Appendix D for details.

Object Motion

Object motion involves the positional displacement of CAD objects during a simulation. This feature allows a user to simulate objects on a conveyor and similar mechanisms.

Object motion proceeds as follows: first the CAD object to be moved during a simulation must be created. Then, a motion planner for the same object should be created. During the simulation, a series of events for each interpolation interval is generated until the object reaches the desired target position. In each event, the position, according to the current interpolation time, is calculated. At the end of the simulation, the simulation engine's list of motion planners is deleted.

In more detailed manner, the whole process can be presented through following steps:

- Through a Workspace Simulation Language program the user calls *SetObjectNextTarget* (in the *CadObject* class). The object speed can be changed by specifying the *value* parameter in *SetObjectSpeed* function.
- *SetObjectNextTarget* then makes a new simulation event called *OnObjectMove* and adds it to the simulation engine.
- *OnObjectMove* event first checks whether the target was reached (if so, a zero is returned). While moving to a target, a series of simulation events is generated for each simulation interval until the object reaches the target. Hence, it is checked whether the current simulation event is the first event in the list. If it is a new default motion planner is created. Then the initial position of the object is set to its current CAD position. The position of the target point is then added to the motion planner.
- The new position of the CAD object is determined with *GetNextObjectStep* (defined in *OnObjectMove* event). This is quite similar to the Realistic Robot Simulation specified *GetNextStep*. However, the only parameters that this function takes are the location and the elapsed time. If the desired target has been reached, a non-zero value is returned, otherwise the returned value will be zero.
- On the first interval the Cartesian distance, the total time and the amount of orientation change are calculated (Figure 5.8). The motion of the object is just simple

linear motion, that is, the object simply moves along a straight line from its start position. The time is determined from the distance and the specified velocity. The object is moved at a constant velocity. The change in orientation is determined by taking the difference of the start and end angles after they have both been converted to roll, pitch, and yaw angle form.

- On the first, and any interval after, the trajectory of the object along a linear path is calculated. The object's base frame will follow the calculated trajectory. The position that object needs to reach at the end of the simulation interval is determined by:
- Finding the vector to move along (between the target and start vectors).
- Normalizing the new vector.
- Scaling the unit vector to find the new position and orientation (for each of the orientation angles: roll, pitch and yaw) of the object while moving along the direction of the that vector:

$$NewObjectPosition = OldObjecPosition + TotalDistance \cdot \frac{SimulationIntervalDuration}{TotalMotionTime}$$

$$NewObjectOrientation = OldObjectOrientation \cdot TotalAngularChange \cdot \frac{SimInterDuration}{TotalMotionTime}$$

- After that, the object starts its motion.
- If the target position has not been reached, a new simulation event will be generated.

In order to make the simulation more accurate, input values to the robot controller must be taken into account. Therefore, the next section (5.4) describes the creation and the main characteristics of the simulation preprocessor.

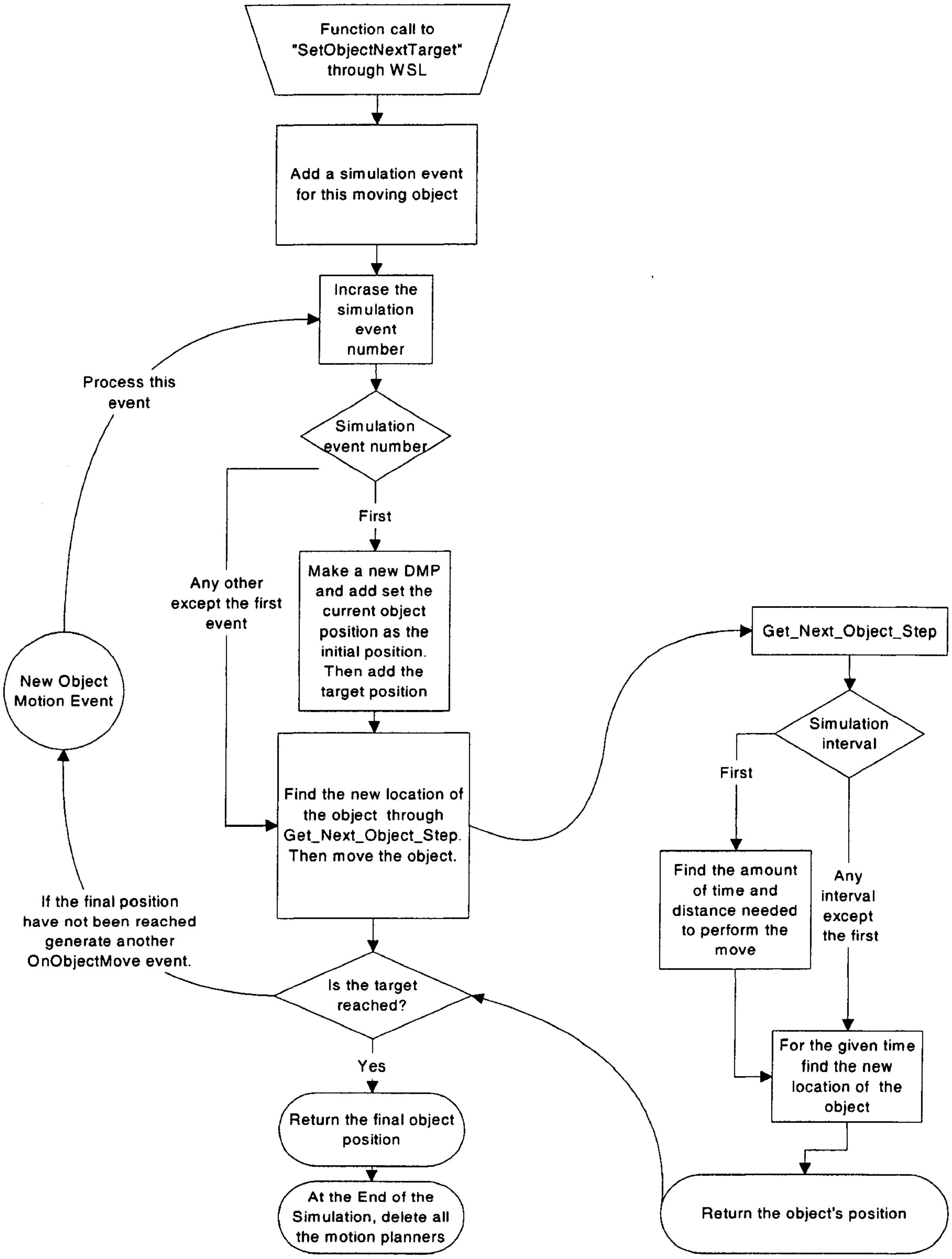

Function call to "SetObjectNextTarget" through WSL
Add a simulation event for this moving object
Incrase the simulation event number
Process this event
Simulation event number
First
Any other except the first event
Make a new DMP and add set the current object position as the initial position. Then add the target position
New Object Motion Event
Get_Next_Object_Step
Simulation interval
First
Find the new location of the object through Get_Next_Object_Step. Then move the object.
Find the amount of time and distance needed to perform the move
Any interval except the first
If the final position have not been reached generate another OnObjectMove event.
Is the target reached?
For the given time find the new location of the object
Yes
Return the final object position
At the End of the Simulation, delete all the motion planners
Return the object's position

Figure 5.8. Object motion algorithm (its implementation allows CAD objects to be moved during the simulation).

5.4. CREATION OF THE G-CODE LANGUAGE PREPROCESSOR

The process of CNC programming has three basic stages:

- Stage 1: Pre-processing.
- Stage 2: Uploading or creating a CNC program (in G-code language).
- Stage 3: Program execution.

In order to accurately simulate the work of a CNC controller, a user must be provided an interface to define all the machine specific parameters stored in a real controller. Having G-code part programs as the only input information is not enough for the simulation software to operate correctly, because some G-code instructions use the parameters that are defined and stored in a controller only, without any data exposure in the program listing. For instance, G55 is a G-code instruction that specifies a reference work coordinate frame, but the position and orientation of that frame is stored in a controller only, whilst part program contains just the reference to an address in controller's memory.

The machine data preprocessing involves several activities, such as:

- Placing the tools into the tool turret (tool magazine).
- Machine table data input.
- Setting the system variables, etc.

Most of the contemporary CNC machines can store more than one tool into a tool turret. The tools are placed in the magazine slots manually by the CNC operator or automatically by manipulator arm. Each position in the magazine is defined by its unique ID, which is known to the machine controller. So, when a command addresses a specific tool, the controller activates the magazine, and brings the requested tool in the loading position.

The next step involves entering the tool offsets into a machine's controller. This action has to be performed every time the tools in the magazine change. Therefore, the tool parameters have to be changed. The tool parameters depend on the machining processes – turning, milling, drilling, grinding, etc. Some examples of machining operations and related tool parameters are as follows:

- Milling – tool diameter and height.
- Drilling – drill's length and diameter.
- Grinding – grinder's diameter, width, etc.

The tool offset values are stored in table format (often referred to as a "machine table"). The data is stored into the controller's memory and is called from the CNC program.

Tool offsets are not the only data stored in machine tables. Frame offsets, fixed points, reference points, variable declarations, command aliases, etc. are stored in machined tables as well. That statement imposes the existence of several machine tables dedicated to different parameters. It is important to know that all of the mentioned parameters are entered before the machining process begins. More detailed explanation of the mentioned parameters follows:

- *Offset frames* - Allen-Bradley 9 series controllers allow predefinition of up to 99 work coordinate frames. Work coordinate frame is the reference coordinate system that can be arbitrarily located within machine's workspace depending on the nature and the complexity of the manufacturing task. To predefine a frame means that the position of a work coordinate frame's origin, with respect to machine's coordinate frame, can be saved to a machine table and retrieved during the program execution, if the appropriate G-code instruction was used.
- *Tool offsets* – These offsets specify the diameters of waterjet or abrasivejet streams as well as tool lengths.
- *Fixed positions*, as well as *Reference Points* are the points within machine's workspace that are significant for a certain manufacturing task. For example, security approach and depart positions located in such a way to ensure that machine will not collide with any obstacles during work, can be predefined and referenced during the program execution.
- *Aliases* – G-code (or CNC code) is composed of "words". A word usually consists of a letter and a number. For example: M30, G111, N10, etc. Each word represents a

specific command, which moves the machine, or performs some other activity such as turning the waterjet on/off, abrasive on/off, rotating the spindle in clockwise or counter clockwise direction, etc. It is possible to set an alias for most of the commands, and later to uses aliases in the program. So, instead of M8, a command can be called as a *WaterJetOn*, for example. Aliases are stored in a machine table as well.

- *Variables* – just like the aliases are stored in a machine table before the machining begins, the same principle applies to variables. They are declared and stored in a CNC controller and they can be referred to once the machining begins.
- *Security Zones* – security zones represent a two-dimensional area or three-dimensional space that must not to be entered by a tool. Security zones are defined by specifying the lower and upper limits, i.e. Xmin, Xmax, Ymin, Ymax, Zmin, and Zmax.
- *Working Zone* – somewhat similar to Security Zones. Working zone is the zone in which the tool center point moves. If the tool center point position is out of the Working Zone, the motion interruption occurs automatically.
- *Home Positions* – besides the primary home position, secondary and tertiary home positions can be specified.

Visual Basic for Applications Language Preprocessing Form

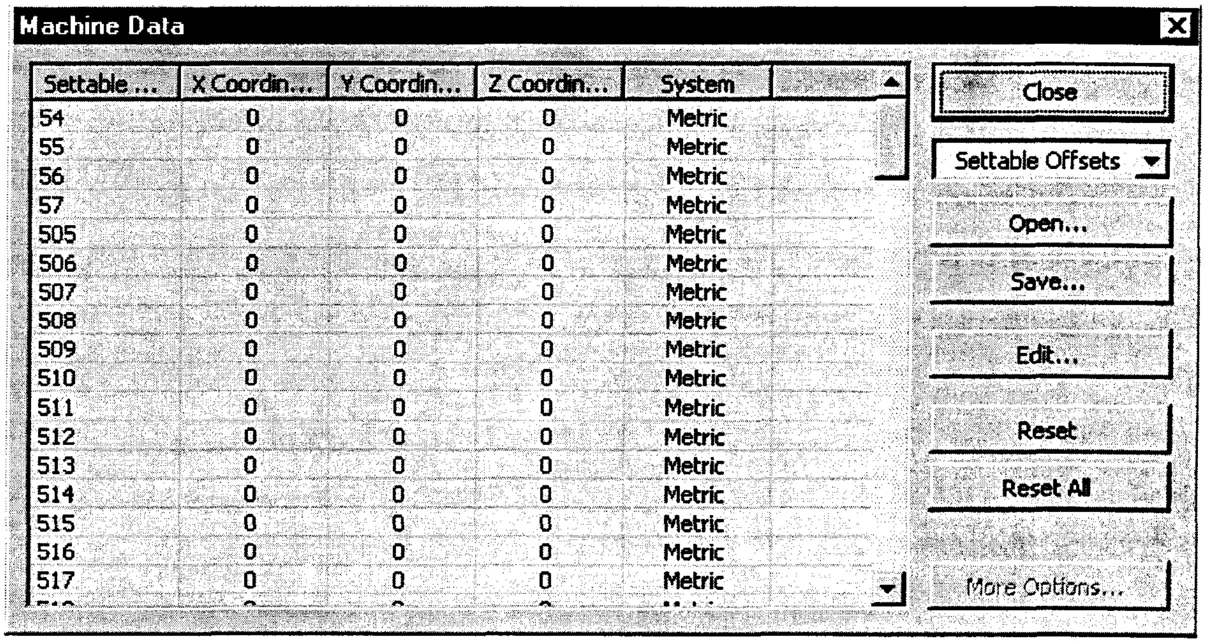
The next section will describe the preprocessing form created in Visual Basic for Application language that takes into account all the machine data parameters explained above.

How to access the form

The form shows up after a user starts the simulation of a 5-axis CNC machine and if the chosen manufacturing operation is waterjet cutting. A message box pops up giving a user an option to fill in the controller parameters.

The preprocessing form

The main purpose of this form is to provide a user with a simple, yet effective way of storing and editing machine data (Figure 5.9).



Settable ...	X Coordin...	Y Coordin...	Z Coordin...	System	
54	0	0	0	Metric	
55	0	0	0	Metric	
56	0	0	0	Metric	
57	0	0	0	Metric	
505	0	0	0	Metric	
506	0	0	0	Metric	
507	0	0	0	Metric	
508	0	0	0	Metric	
509	0	0	0	Metric	
510	0	0	0	Metric	
511	0	0	0	Metric	
512	0	0	0	Metric	
513	0	0	0	Metric	
514	0	0	0	Metric	
515	0	0	0	Metric	
516	0	0	0	Metric	
517	0	0	0	Metric	

Figure 5.9. Visual Basic for Applications language preprocessing form shows the list of settable offsets. The form appearance changes depending on the option chosen from the drop down list.

Main drop-down list

A user can set the following data (Figure 5.10):

- Tool and frame offsets,
- Reference points,
- Home positions,
- Fixed positions,
- Variables and aliases.

Which parameters will be seen in the table depends on the item selected in the drop-down list.

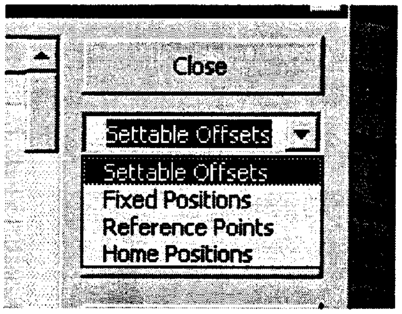


Figure 5.10. Expanded drop-down list that shows possible machine data groups that can be predefined.

Open

It is quite convenient to store the offsets permanently and then to upload/save them onto a disk. This feature is provided with *Open* and *Save* buttons.

By clicking on *Open* button, a standard Windows open dialog box pops up, and a user can select a file to upload. The file is in plain ASCII format, therefore the extension is *.txt.

The file will contain the values of all the items in the drop list. The file format is the following:

- A file must have NCFILE keyword in the first line.
- The second line contains OFFSET.
- Data lines are described as follows:
  - Offset ID,
  - White space(s)
  - Offset in X direction,
  - White space(s)
  - Offset in Y direction,
  - White space(s)
  - Offset in Z direction.
- The same applies for fixed positions as well.

- The file ends with ENDFILE.

For example:

```
NCFILE
OFFSET
54      200.0001                200.0002        200
55      255                       255           255
56      100                       200           300.001
57      0                         0             0
...
FIXED
 1      0                         0             0
 2      0                         0             0
ENDFILE
```

Example 5.1. Sample data file containing three preset frame offsets all the default values for all the other items.

Save

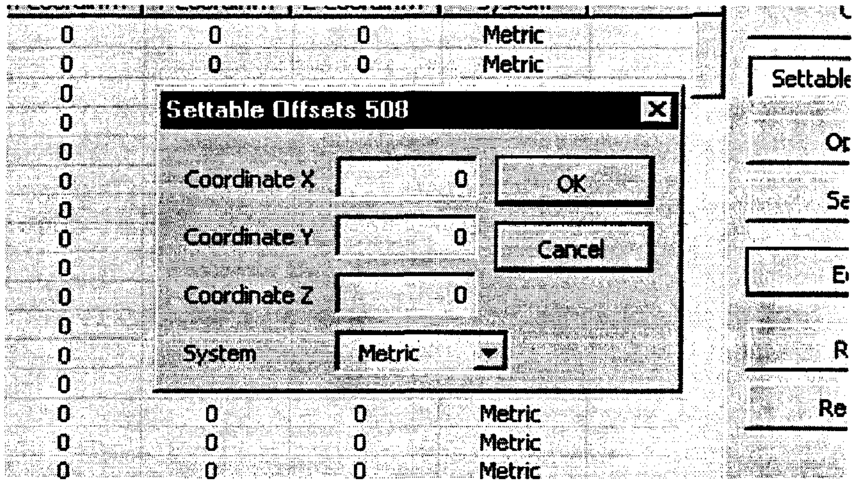
After clicking on this button, the standard Windows *Save* dialog pops-up, and enables the user to save the data into a file. Format of the data saved is shown in the previous example, and the file extension is *.txt.

Edit

There are two ways to edit the data shown in the table:

- Double-click on the table,
- Select an item in the list, and click *Edit* button.

A new form pops up (Figure 5.11), enabling the user to change the data (X, Y, Z) and the type of the measurement system in which the X, Y, Z are expressed.



Settable Offsets 508

Coordinate X: 0

Coordinate Y: 0

Coordinate Z: 0

System: Metric

0	0	0	Metric
0	0	0	Metric
0			
0			
0			
0			
0			
0			
0			
0			
0			
0			
0			
0	0	0	Metric
0	0	0	Metric
0	0	0	Metric

Figure 5.11. Edit dialog box that shows how the parameter values can be changed. Edit dialog box changes its appearance depending on the machine data to be edited.

Reset

By selecting an item in the list and clicking on the *Reset* button, X, Y, and Z values are set to zero.

Reset All

By clicking on the *Reset All* button the values of all the items in the list are set to zero.

More Options

This button has not been activated yet. In case that some additional features have to be customized, this button can be used to call another form, or to extend the height of the existing form, providing more data to be edited.

CHAPTER VI

OFF-LINE PROGRAMMING MODULE DEVELOPMENT

6.1. OFF-LINE PROGRAMMING IN ROBOTICS

Present teach methods of programming industrial robots have proved to be satisfactory where the proportion of teaching time to production time is small, and also when the complexity of the application is not too demanding. They involve either driving a robot to required positions with a teach pendant or physically positioning the robot, usually by means of a teach arm. Teach methods as such necessitate the use of the actual robot for programming.

Off-line programming may be considered as the process by which robot programs are developed, partially or completely, without requiring the use of the robot itself. This includes generating point coordinate data, function data, and cycle logic. Developments in robot technology, both hardware and software, are making off-line programming techniques more feasible. These developments include greater sophistication in robot controllers, improved positional accuracy, and the adoption of sensor technology. There is currently considerable activity in off-line programming methods, and these techniques are employed in manufacturing industries (Nof, 1999).

6.1.1. Why Should Off-line Programming Be Used?

Robot on-line programming can be time-consuming. As the robot remains out of production, on-line programming can substantially reduce the utilization of the robot, sometimes to the extent that the economic viability of its introduction is questioned.

Many early robot applications involved mass production processes, such as spot welding in automobile manufacturing lines, where the reprogramming time required was either absent or minimal. However, for robot - manufacturing applications to be feasible in the field of small and medium batch production, where the programming times can be substantial, an off-line programming system is essential. The increasing complexity of robot applications, particularly with regard to assembly work, makes the advantages associated with off-line programming even more attractive. These advantages may be summarized as follows (Nof, 1999):

1. *Reduction of robot downtime.* The robot can still be in production while its next task is being programmed. This enables the flexibility of the robot to be utilized more effectively and therefore decreases the programming costs.
2. *Removal of programmer from potentially hazardous environments.* As more of the program development is done away from the robot this reduces the time during which the programmer is at risk from aberrant robot behavior.
3. *Single programming system.* The off-line system can be used to program a variety of robots without the need to know the idiosyncrasies of each robot controller.

The main disadvantage of off-line programming systems is that they generally show some differences between the desired and the real motion after having been uploaded to the robot controller. Individual deviations include those from the nominal position and orientation of the end-effector, from the nominal path and path velocity, as well as from the nominal cycle time (refer to section 1.1.).

6.1.2. Requirements of an Off-line programming system

Different off-line programming systems employ different approaches to the programming method. Yet, despite their differences, they contain certain common features essential for off-line programming. The following list gives the requirements that have been identified to be important for a successful off-line programming system (Nof, 1999):

1. A three-dimensional world model, that is, data on the geometric descriptions of components and their relationships within the workplace.
2. Knowledge of the process or task to be programmed.

3. Knowledge of robot geometry, kinematics (including joint constraints and velocity profiles), and dynamics.
4. A computer-based system or method for programming the robots, utilizing data from items 1, 2, and 3. Such a system could be graphically or textually based.
5. Verifications of programs produced by item 4. For example, checking for robot joint constraint violations and collision detection within the workplace.
6. Appropriate interfacing to allow communication of control data from the off-line system to various robot controllers. The choice of a robot with a suitable controller (i.e. one that is able to accept data generated off-line) will facilitate interfacing.
7. Effective man-machine interface. Implicit in off-line programming is the removal of the programmer from the robot. To allow the effective transfer of his skills to a computer-based off-line system, it is crucial that a user-friendly programming interface be incorporated.

To fulfil the fourth requirement of the above numbered list, a graphical computer-based system for off-line programming of 5-axis waterjet cutting gantry robot has been developed and will be presented in the next section.

6.2. CREATION OF OFF-LINE PROGRAMMING MODULE FOR FIVE-AXIS WATERJET CUTTING GANTRY ROBOT

In order to create the part programs from graphical path representations in Workspace the following tasks have been done:

1. The CAD model of a test part has been designed (as explained in section 4.2).
2. The robot graphical path has been automatically created (using Workspace's Automatic Path Generation module) and the path properties set.
3. The off-line programming software module has been written to convert the graphical path properties into G-code part programs.

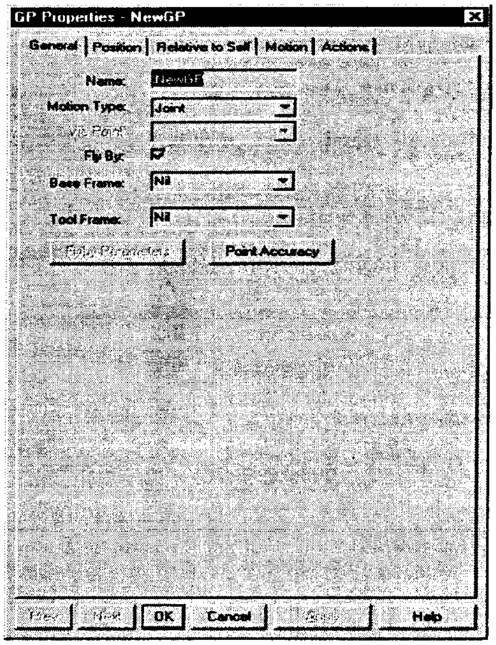
The terms: *path, automatic path generation,* and *geometric points* are important for understanding of following paragraphs, therefore short definitions of those terms are provided below.

A *path* is a list of all the geometric points (GPs) that a robot is to follow during a sequence of motions. It is displayed graphically as a series of lines linking together the geometric points with direction arrows showing the direction of the motion.

Automatic path generation allows the user to automatically generate a path on a CAD model of a workpiece. The automatic path takes into account the parameters set by the user, including boundaries, holes, material, thickness and speed, and generate a suitable path that the robot will follow.

Robot teach points are called *geometric points* in Workspace. Geometric points are robot endpoint target locations that may be saved with a model and which may be manipulated as though they were objects. They not only do represent the position and orientation of a robot target position, but also store attributes describing the nature of the motion to be used when travelling towards target geometric point from current geometric point on the same path, as well as attributes describing the actions to be performed when the GP is reached. Every GP along the path contains an editable set of attributes. For waterjet cutting application, GPs have the following properties (Figure 6.1):

- Motion Type (Rapid, Linear, Circular),
- Feedrate value,
- Acceleration value,
- Position parameters,
- Orientation parameters,
- Dwelling time,
- Waterjet stream activated (an action), and
- Waterjet stream deactivated (an action).



Name: NewGP
Motion Type: Joint
Via Point:
Fly By: ☑
Base Frame: Nil
Tool Frame: Nil

Figure 6.1 Geometric point properties dialog box. This figure shows just one tab-window of the dialog box. Different tab-window contain different sets of geometric point properties.

In order to send the programs developed off-line in Workspace to a robot, they must first be translated from the internal graphical representation (a *path*) into a robot program written in the selected robot language (a *track*). This process is called *Path to Track* translation. Figures 6.2 and 6.3 display an arbitrary created graphical path in Workspace and the corresponding G-code part program based on the captured path properties. When *Write Program* is selected from the context menu for a path, the *Path to Track* class for the robot language of the robot to which the path belongs is instantiated, and its *WritePath* method executed.

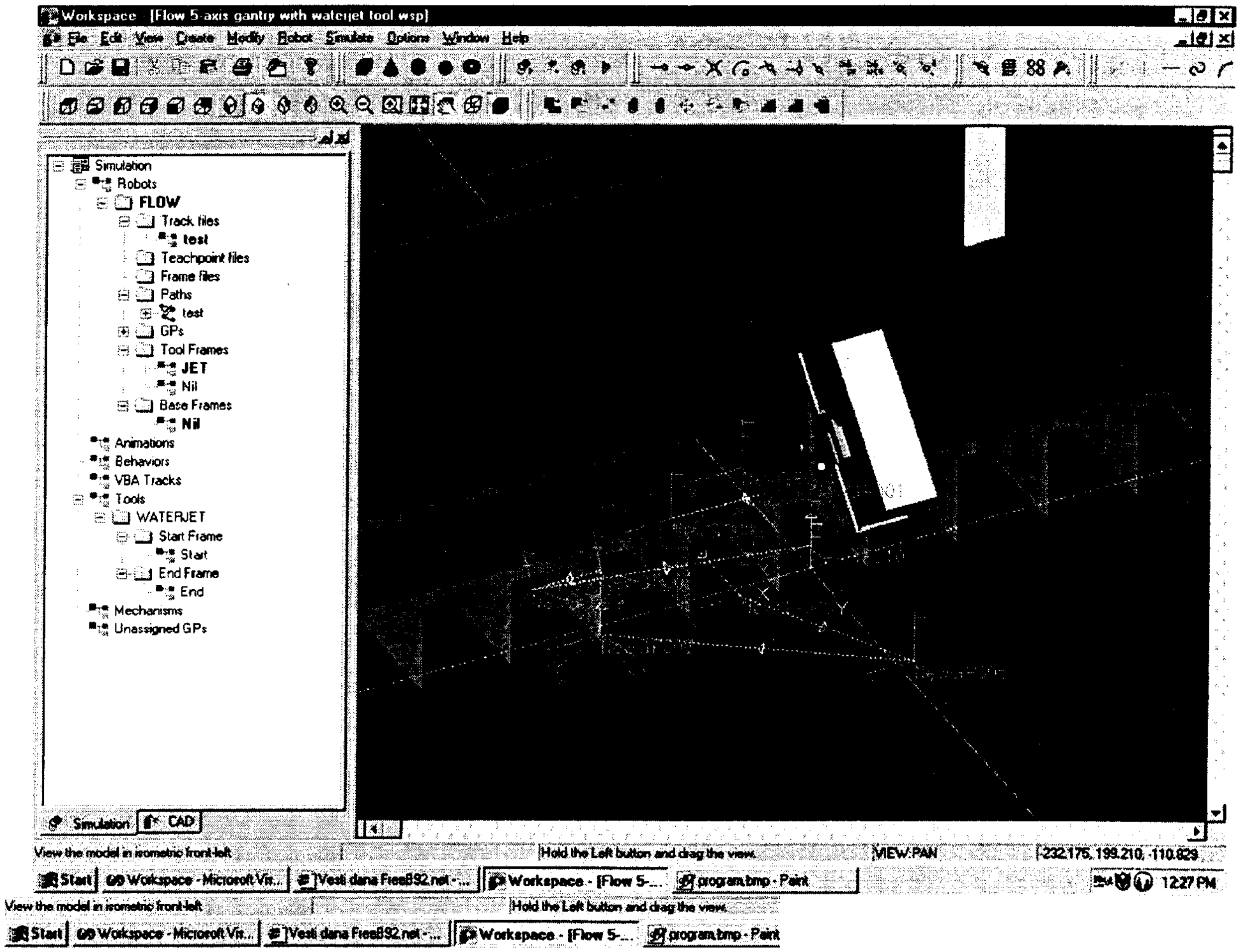
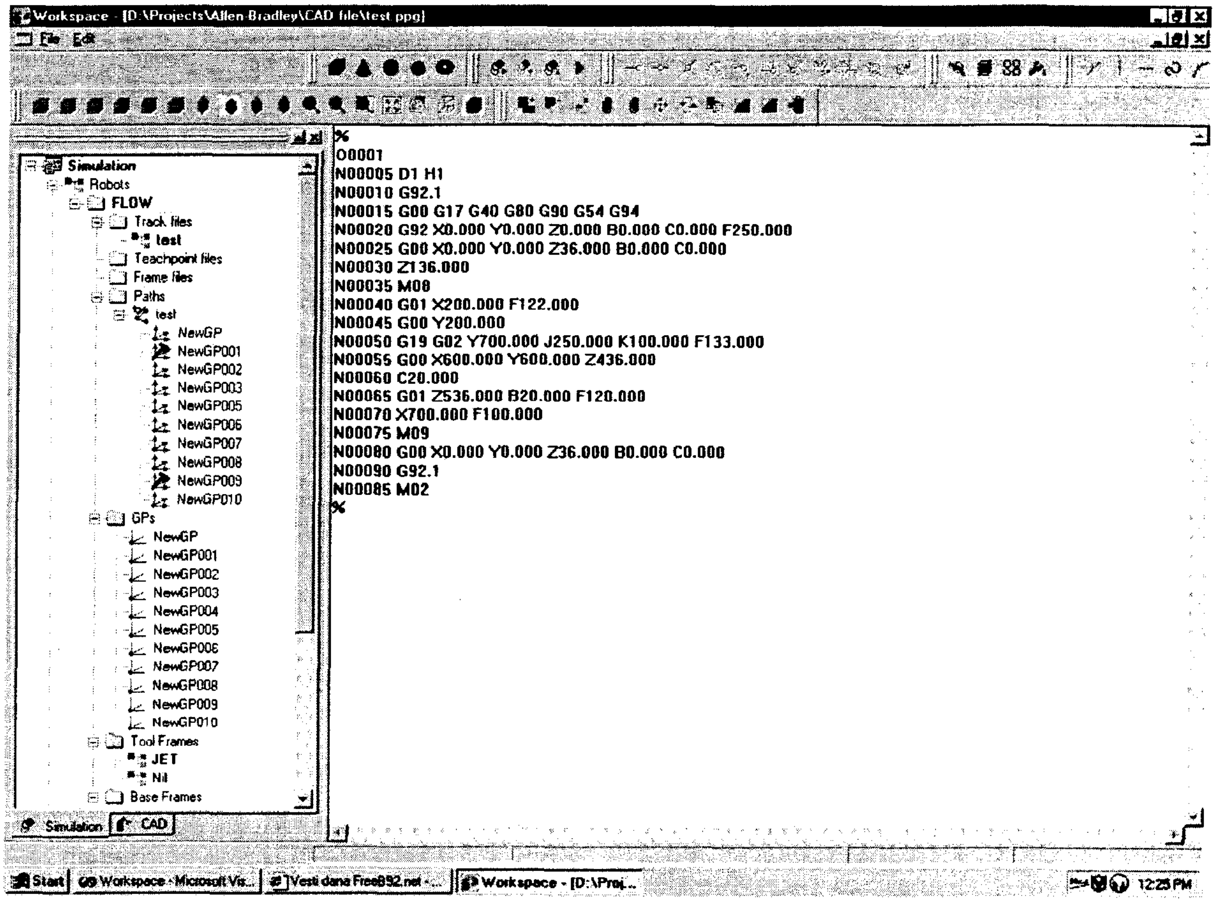


Figure 6.2 A graphical path representation in Workspace. Series of geometric points are connected with line segments, while arrows show the direction of motion.

A root path may contain one or more sub-paths. Each path, including the root path, contains a list of one or more *pathtarget* pointers. (It's logically possible for a path to contain zero *pathtargets*, but such a path is not relevant for off-line programming.). A *pathtarget* can be either a *GPMove* or another *Path* (as shown in example 6.1). The paths containing cycles, wherein one path contains a reference to one of its ancestors, are not considered well formed.



```
%
O0001
N00005 D1 H1
N00010 G92.1
N00015 G00 G17 G40 G80 G90 G54 G94
N00020 G92 X0.000 Y0.000 Z0.000 B0.000 C0.000 F250.000
N00025 G00 X0.000 Y0.000 Z36.000 B0.000 C0.000
N00030 Z136.000
N00035 M08
N00040 G01 X200.000 F122.000
N00045 G00 Y200.000
N00050 G19 G02 Y700.000 J250.000 K100.000 F133.000
N00055 G00 X600.000 Y600.000 Z436.000
N00060 C20.000
N00065 G01 Z536.000 B20.000 F120.000
N00070 X700.000 F100.000
N00075 M09
N00080 G00 X0.000 Y0.000 Z36.000 B0.000 C0.000
N00090 G92.1
N00085 M02
%
```

Figure 6.3. The part program written in G-code language based on the graphical path shown on the Figure 6.2.

MainPath		SubPath1
GP1		
GP2		
GP3 ViaPoint1		
SubPath1	-----→	GP4
GP7		GP5
GP8		GP6
GP9		

Example 6.1. Paths in Workspace are composed of series of targets (*pathtarget* pointers) that can be either geometric points or subpaths. Subpath nesting is allowed to be eight levels deep.

In general, *Path to Track* translation requires less development effort than *Track to Path* translation (the process of reading a robot language track file and producing a graphical path during simulation). This is because *Track to Path* must process every valid program in a given robot language, including statements that aren't currently supported by off-line programming module. *Path to Track* needs only to be concerned with emitting a robot language program (track) which captures the intent of a graphical path. This tends to be a small subset of the robot language. Only three actions need to be provided for:

- *Motion.* This includes definition of teachpoints, and specification of a path supported motion parameters (motion type, speed, acceleration, via point/centre for circular moves, dwelling (idle) time).
- *Sub-program calls.* This entails creation of separate tracks for each sub-program.
- *Tool actions* which have a standard implementation in the target language (such as M08 and M09 in G-code language), and which have a standard representation in graphical paths (such as *WaterjetOn* and *WaterjetOff*).

G-code language does not separate the teach point data from the move commands.

Instead, the move command will contain all the geometric data required to interpret the move. One consequence of this is that no initial tree search to enumerate the teach points is required. Another consequence is that a related process of updating the teach point locations, instead of completely rewriting the program file, when a user decides to change the position and orientation properties of teach points in the path, is not possible.

Also, G-code language provides for motion parameters, which haven't changed from the previous move to be omitted from the program. Therefore, a solution has been found for the storage and retention of the relevant state data (this includes all motion parameters) from one move to the next. The state data must also be reset at the appropriate times (at least for each new root path; possibly more frequently).

Knowing all the previously mentioned path parameters and taking into account the grammar of the G-code programming language, the C++ source code has been written to recast all the path data into sequence of CNC machine instructions. The source code contains a super class named *CPathToLanguage* and a derived class *CPathToGCode.* The source code for the class *CPathToGCode* is presented in Appendix I. Three most important methods (member functions) of the derived class, which will be explained in the following paragraphs, are:

- *WritePath,*
- *WriteFunction, and*
- *WriteMove.*

All three of these functions have virtual prototypes in the *CPathToLanguage* class, which is the super class of *CPathToGCode* class. The way how these three methods communicate is depicted on the Figure 6.4.

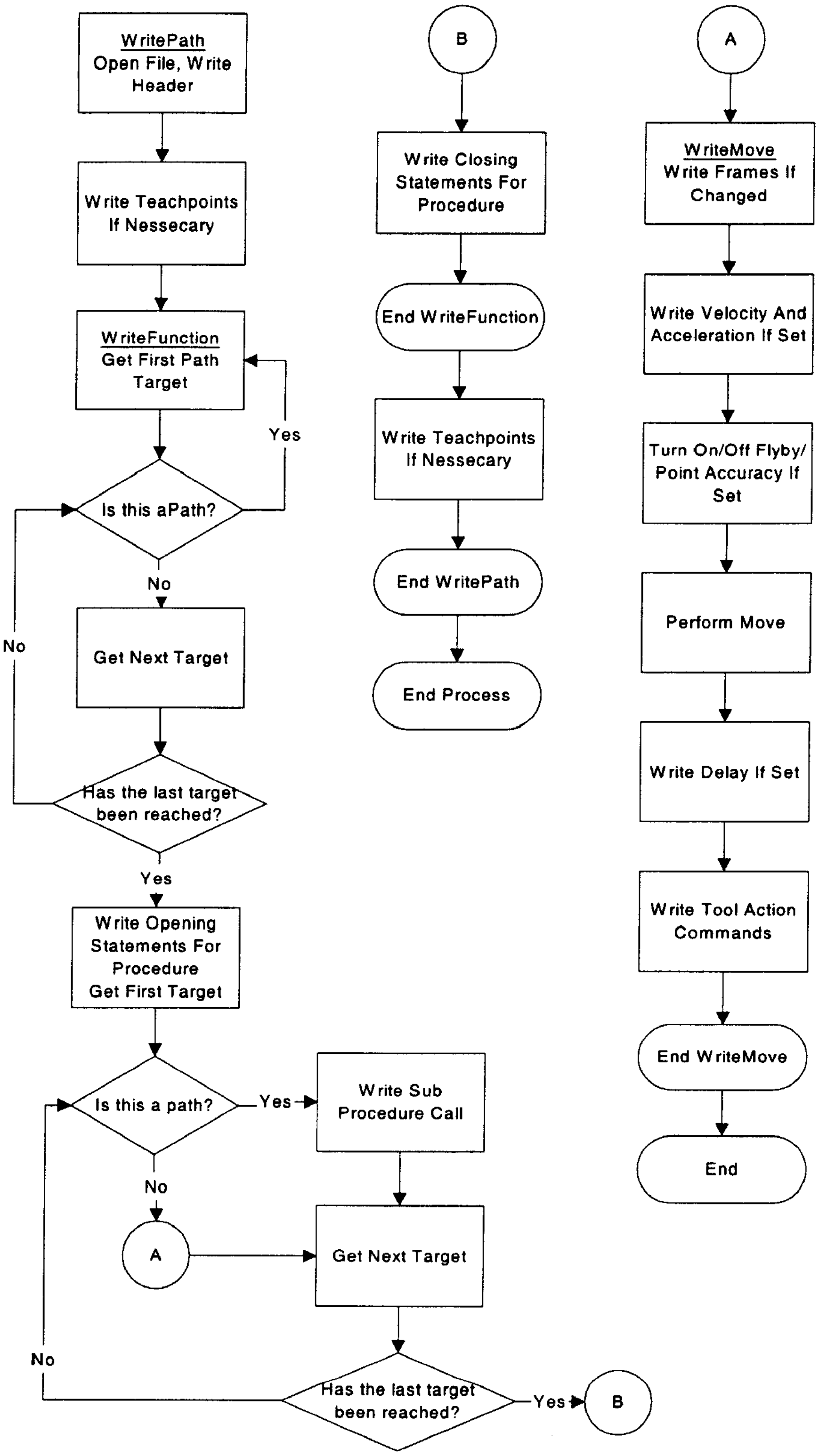


Figure 6.4. The function call flow among the three most important methods of the *CPathToGCode* class: *WritePath*, *WriteFunction* and *WriteMove*.

The constructor of this class is used to set up the default values of any member variables needed for the language translation. For instance, *m_strLanugage* is a string variable set to the name of the language being produced.

The *WritePath* function is the main function of the translation class. This function is called by Workspace and carries out the translation by calling other class member functions. The first action that this function does is to open the output file for writing using the member *m_strFileName* for the file name. After this the *WritePath* function writes strings to the file, and calls other translation member functions.

The *WriteFunction* procedure performs two actions:

- Firstly, it determines how many sub paths are embedded in the main path by calling itself recursively, and
- Secondly, it loops through the counted paths, starting with the deepest embedded subpath and proceeding up, and writes out the formatted path properties into the output file based on the syntax of the target language.

The *WriteMove* function performs the task of writing out the movement, acceleration, velocity, frame, delay, and tool action commands. This is done by capturing the values of the various *GPMove* properties and by outputting the lines of code based on which properties have been set.

Emitted G-code part programs have been beta-tested directly on gantry robots in Flow Robotics Company, Jeffersonville, IN.

6.3. ON-LINE TESTING

The purpose of the on-line testing was to determine positional accuracy of the teach points written in a part program created by the off-line programming module and cycle time accuracy of the simulated manufacturing task.

The CNC machine used for testing was an "AF-series 5-Axis Three Dimensional Shapecutting Machine" equipped with Allen Bradley 9/260 controller and manufactured by Flow Robotics Company. The complete technical specification of the CNC machine is presented in Appendix C.

The geometry of the testing part, the creation of its CAD model in Workspace and the reasons why that particular part has been chosen to be tested have been explained in section 4.2.

The testing procedure was organized in the following way:

1. The material used for cutting was a plate-shaped rubberized compressed foam. This material is very cheap and is used primarily for preliminary testing when the roughness of the surfaces to be cut is not of primary importance. Due to very high water jet pressure (up to 415 MPa) and very low rigidity of the material to cut through, the edges of the part after manufacturing were visibly rough (the material was bending under the force produced by water jet). If steel or aluminum had been used instead of compressed foam, the edges would have been much smoother and within the tolerances achievable by waterjet cutting process (0.075 mm).
2. The robotic cell has been set up in Workspace, as explained in Chapter IV.
3. The corresponding part program has been created off-line in Workspace (the listing of that part program is presented in Appendix J), as explained in Chapter VI.
4. The personal computer has been interfaced with the robot controller via RS-232 serial interface cable. The part program has been downloaded from the personal computer and uploaded to the robot controller using Allen-Bradley Off-line Development System (ODS) software, previously installed to the personal computer.
5. The part has been manufactured and the cycle time recorded from the controller's display.
6. The manufacturing task has been simulated in Workspace.

The robot controller language accepts only axis values as parameters of teach point definitions (values of X, Y, Z, B, and C axes). Stated differently, teach points cannot be defined in Cartesian values (X, Y, Z, Roll (rotation about Z axis), Pitch (rotation about Y axis), and Yaw (rotation about X axis)) with respect to the machine coordinate frame. Therefore, due to the fact that the robot controller cannot perform inverse kinematics transformation, forward kinematics accuracy has been tested only. Workspace teach-pendant values (both Joint and Cartesian) of the teachpoints created during simulation of the task, have been compared with Joint and Cartesian values read from the robot controller. The notified difference between the Cartesian values read from the controller and Workspace has been within the range of 1 to 3 thousands of a millimeter. Taken into account that the machine accuracy is 0.127 mm the recorded difference was obviously satisfactory.

Simulation and real robot cycle time accuracy has been determined comparing the corresponding cycle time values recorded in Workspace and in the robot controller. The determined ratio was 126.9s (simulation) / 129.7s (real time), therefore the error margin of 2.16% has been identified. The cycle time accuracy has been assessed as satisfactory and acceptable, because the error value was below the acceptable upper limit of 3%, set by Realistic Robot Simulation standard. The possible sources that caused that error margin have been listed in section 1.1, while Realistic Robot Simulation concept has been explained in sections 2.3 and 5.2.2.1.

CHAPTER VII

CONCLUSION

This dissertation aims to prove that the use of Realistic Robot Simulation interface in conjunction with PRECCX parser generator in robotics simulation and off-line programming software packages can substantially improve the positional and cycle time accuracy of a 5-axes waterjet cutting gantry robot.

In order to verify and validate the quality of the written software modules, a 5-axes gantry robot manufactured by Flow Robotics Company and equipped with an Allen-Bradley 9 series controller has been used for testing. The mentioned simulation and off-line programming software modules are not stand alone applications, therefore they are dependent on the software platform they are written for. In this case, Workspace 5® software package serves the purpose of being the platform for the written add-ins. g

The dissertation is organized as follows:

- ***CAD modeling of the workpiece and the 5-axis gantry robot.*** Kinematics definitions of the CAD models.
- ***Development of the simulation module.*** This is the central and the most complex part of the dissertation. The first sub-task is the development of G-code (CNC controller programming language) language translator. Compiler-compiler software utilities Flex and PRECCX (Prettier Compiler-Compiler Extended) have been used for lexical analysis and parsing of the G-code language. Parser action statements have been written to emit Workspace Simulation Language (Visual Basic for Applications based simulation language, which contains calls to the functions of the motion planner via Component Object Model RRS interface) code. Also, all the Realistic Robot Simulation services called from the parser's action statements have been designed and implemented in the motion planner module of the simulation platform software. The preprocessing graphical interface has been developed in order to capture the internally stored robot controller information that is not provided in the part programs, but presents a mandatory input for accurate simulation.
- ***Development of G-code off-line programming module.*** Knowing all the path parameters defined during simulation and taking into account the grammar of the G-

code programming language, the C++ source code has been written to recast all the path data into sequence of CNC machine instructions.

The flow chart on Figure 7.1 shows the interactions among the modules written by the master's candidate and the modules of the software package built-in the development platform. Different background colors represent different levels of participation (color mappings are contained within the figure caption).

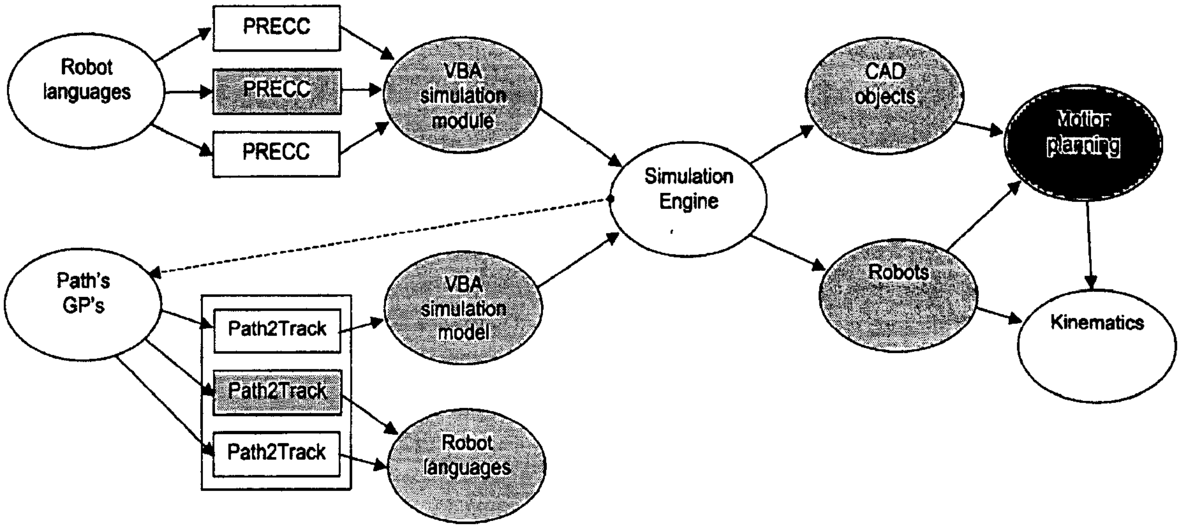


Figure 7.1. Simulation and off-line programming modular concept of simulation platform software, where different colors represent different levels of author's participation (white – didn't participate, light gray – input modules, gray – complete design and implementation, dark gray – partial design and implementation)

Relevance of the issue of this thesis will be presented through the next three major points:

- ***Highly accurate simulation due to implementation of Realistic Robot Simulation services.*** Strictly following the standardization rules in the Realistic Robot Simulation specification default set of Realistic Robot Simulation services has been implemented in default motion planner of the simulation software. In other words, when robot languages which are developed by manufacturers who did not participate in Realistic

Robot Simulation consortium (including the CNC languages) need to be simulated, the default set of Realistic Robot Simulation services will be called to provide the accurate motion of the simulated robot. The Realistic Robot Simulation interface has been tested on software and hardware platforms used for robotic simulation in the automotive industry and has demonstrated impressive results of accurate simulation of motion behavior, robot kinematics, and condition handling. It has been proven that the deviation between simulated and real joint values is less than 0.001 radians. The ideal case, of course, would be if there was not any difference between the simulated and real joint values. However, taking into account that non-RRS simulation software packages have on average approximately 10 times lower angular accuracy, Realistic Robot Simulation interface showed considerable improvement. Concerning task cycle times, a difference of less than 3% could be reached. Again, in comparison with the non-Realistic Robot Simulation systems where the cycle time difference is in range of 5 to 10%, Realistic Robot Simulation systems are obviously better, but their cycle time accuracy can still be significantly improved (Realistic Robot Simulation II interface promises 99.5% cycle time accuracy and 99.9% joint accuracy, which remains to be proven).

- ***The first industrial application of currently the most efficient compiler-compiler utility - PRECCX.*** According to the conducted literature and Internet Web search, this will be the first industrial application of PRECCX software utility. Making a compiler by using some of the contemporary programming languages (mainly C and C++) is attainable, but it takes too much time. Alternatively, using PRECCX for the same purpose will be faster and less prone to errors and also, it will have some advantages over the most frequently used compiler-compiler utilities –YACC and Bison.
- ***CNC controller simulation based on the customization of the robotic simulator.*** CNC machines can be considered gantry robots as far as their kinematics is concerned, but programming language that they use (often called G-code language) has its own characteristics that differ from the ones of the robot programming languages. Typical example for that is the definition of circular motion. Therefore, using Realistic Robot Simulation interface to simulate CNC programming language

has required considerable amount of work, in depth knowledge of CNC controller software and Realistic Robot Simulation specification. Mapping CNC to default Realistic Robot Simulation instruction did not necessarily have one to one matching due to limited compatibility between the interface and the language.

This thesis had its practical verification in Flow International Corporation. The results of preliminary testing showed the following:

- When the position and orientation values (both Joint and Cartesian) of the teachpoints created during simulation of the task, have been compared with Joint and Cartesian values read from the robot controller, the notified difference has been within the range of 1 to 3 thousands of a millimeter. Taken into account that the machine accuracy is 0.127 mm the recorded value difference was satisfactory.
- Simulation and real robot cycle time accuracy has been determined comparing the corresponding cycle time values recorded in Workspace and in the robot controller. The error margin of 2.16% has been notified. The cycle time accuracy has been assessed as satisfactory and acceptable, because the error value was below the acceptable upper limit of 3%, set by Realistic Robot Simulation standard.

From the user point of view, very important feature is that the graphical user interfaces, for both simulation and off-line programming module, are user friendly and very intuitive. User time spent on setting the parameters for those two modules is short, robotics skill level required from is not high, however the output is still accurate and dependable.

Future work on this thesis should be based on the implementation of RRS II services in the motion planner (once they become available to developers), which will provide additional quality features to simulation software packages, such as: axis grouping, signaling (I/O), interrupt handling, et cetera. Also, robot path planning and optimization for waterjet cutting manufacturing application is the area that requires substantial engineering and programming knowledge, and that can significantly complement and upgrade the quality of the presented work.

APPENDIX A: USING PRECCX

Basics

```
@            a =    b c d
@            |      b c
```

The previous notation says that the language structure *a* can be specified by either structure *b* followed by structure *c* followed by structure *d*, or by structure *b* followed by structure *c*.

The '|' character in PRECCX means 'OR'.

Every line of PRECCX code must start with an @ symbol, otherwise it gets written directly into the C code without PRECCX converting it.

There is, however a better way of specifying this construct in PRECCX:

```
@       a =     b c [d]
```

In this case the construct *d* is specified as being optional by enclosing it in square brackets.

Take the specification:

```
@       Boring=             <'z'>*
```

The <> around the 'z' cause PRECCX to look for a C style literal token, in this case 'z'. The * after it, causes PRECCX to look for it 0 or more times.

Valid inputs are:

(nothing)

zzz

z

The + character is similar to the * except that it looks one or more times, eg.

@ *StillBoring= <'z'>+*

Valid inputs are:

z

zzzz

Take the following definition:

@ *Identifier = alpha {alpha | numeric } **

This describes a basic form of identifier, the valid inputs would be:

A

A23jdas

In PRECCX, the *{}* construct is used for grouping, so the above definition would be read as:

An Identifier is an alpha followed by zero or more numbers or alphas.

In C the general format is that a semi-colon follows each line of code. This can be written in PRECCX as:

@ *CLine= line_of_code]<';'>[*

The *"] ["* construct means that although a semi-colon is required, it will not be used for anything. This saves computation time and makes more efficient translators.

Here's an example of a C++ comment:

@ *CppComment = <"//"> ?* $*

This will match anything between the “//” and the end of the line as a comment. The *’?’* character tells PRECCX to match anything except the *End of Line* character. The *’$’* character tells PRECCX to match the *End of Line* character.

Errors

```
@   WhileLoop =       <"WHILE"> expression
@                 line_of_code *
@                 <"WEND">
@                 |       <"WHILE"> expression
@                 line_of_code * ! {: printf("While without Wend Error"); :}
```

If the translator can't find the *"WEND"* keyword, the “*!*” character causes PRECCX to flag an error.

Actions

In the above example the version which caused the error had some C code attached. This C code is what is used to do the translation.

```
@       test=   abc     {: printl("d"); :}
@       a =     <'a'>   {: printf("a"); :}
@       b=      <'b'>   {: printf("b"); :}
@       c =     <'c'>   {: printf("c"); :}
```

If we pass the input *" a b c "* to this test construct, we would get the output *"a b c d"* written to the output.

Attributes

PRECCX has the ability to synthesize attributes, e.g.

```
@       Temp =       <'a'> {@ 1 @}
@              |     <'b'> {@ 2 @}
```

This statement would return 1 if it was passed an *'a'* or 2 if it was passed a *'b'*. The value passed back must always be of type *long* and is passed using the { @ @ } construct.

```
long nTemp =0;

@       Test = {TestForA *}\x
@       TesForA =     <'a'> {: nTemp++; :} {@ nTemp @}
```

The above example will return into the attribute *x* the number of *a*'s matched. This is because PRECCX returns the value from the last occurrence of *TestForA*.

```
@       TestForNot = [<"NOT"> {@ 1 @}]\x
```

This will return 1 if the word *"NOT"* is matched, otherwise it will return 0.

Using the values of attributes

```
@       Test = FurtherTest\x  {: printf( "The Value Returned was %d", $x); : }
```

When using the value of an attribute you must always prefix it with a *$*.

Abstract Syntax Trees

Abstract Syntax Trees (AST's) can represent most syntax. This is a method of representing language without being language specific.

Example: A simple expression:

X = A + 1;

```
        =
      /   \
      X     +
         /  \
        A    1
```

A function can be represented as follows:

```
Function (a, b, ...);
```

```
           / \
 Function    /\
            / \
           /    \
       a        /\
               /  \
              b    ...
```

Abstract Syntax Tree's don't store the language specifics such as the braces around the parameters, the commas between parameters, and the semi-colon after the line of code. If we those specifiers are not stored, a tree can be used to translate into any language.

APPENDIX B: GLOSSARY OF THE PARSING AND LEXICAL ANALYSIS KEY TERMS

Backus-Naur Form

A formal metasyntax (syntax used to describe syntax) used to express context-free grammars.

BNF is one of the most commonly used metasyntactic notations for specifying the syntax of programming languages, command sets, and the like. It is widely used for language descriptions but seldom documented anywhere.

Grammar

A formal definition of the syntactic structure of a language (syntax) normally given in terms of production rules which specify the order of constituents and their sub-constituents in a sentence (a well-formed string in the language). Each rule has a left-hand side symbol naming a syntactic category and a right-hand side, which is a sequence of zero or more symbols. Each symbol may be either a terminal symbol or a non-terminal symbol. A terminal symbol corresponds to one "token" - a part of the sentence with no internal syntactic structure (e.g. an identifier or an operator in a computer language). A non-terminal symbol is the left-hand side of some rule.

One rule is normally designated as the top-level rule, which gives the structure for a whole sentence.

A grammar can be used either to parse a sentence or to generate one. Parsing assigns a terminal syntactic category to each input token and a non-terminal category to each appropriate group of tokens, up to the level of the whole sentence. Parsing is usually preceded by lexical analysis. Generation starts from the top-level rule and chooses one alternative production wherever there is a choice.

Lexical analysis

The first stage of processing a language. The stream of characters making up the source program or other input is read one at a time and grouped into tokens - word-like pieces such as keywords, identifiers, literals and punctuation. The tokens are then passed to the parser.

Parser

An algorithm or program to determine the syntactic structure of a sentence or string of symbols in some language. A parser normally takes as input a sequence of tokens output by a lexical analyzer. It may produce some kind of abstract syntax tree as output.

Parser generator

A program which takes a formal description of a grammar in Backus-Naur Form and outputs source code for a parser which will recognize valid strings obeying that grammar and perform associated actions.

Syntax

The structure of strings in some language. A language's syntax is described by a grammar. For example, the syntax of a binary number could be expressed as

```
binary_number = bit [ binary_number ]

bit = "0" | "1"
```

meaning that a binary number is a bit optionally followed by a binary number and a bit is a literal zero or one digit.

The meaning of the language is given by its semantics.

Token

A minimal lexical unit of a language. Lexical analysis converts strings in a language into a list of tokens. For a programming language these word-like pieces would include keywords, identifiers, literals and punctuation. The tokens are then passed to the parser for syntactic analysis.

APPENDIX C: ALLEN-BRADLEY G-CODE SPECIFICATION

Product Name:	AF-Series 5-Axis Three Dimensional Shapecutting Machine
Product Description:	The Shapecutting cell is a freestanding gantry unit with the working axes located between its vertical support legs and above the work area. The C & B Axis are mechanically arranged so that the cutting tip remains at a constant focal point, which is fixed while the axes are rotating around the fixed point.
Controller:	This machine is electric AC servo motor driven and micro processor controlled from a CNC based Allen-Bradley 9/260.

	Base Axis	Bridge Axis	Vertical Axis
Work Envelope Sizes:	6 Ft Stroke	8 Ft Stroke	2 Ft Stroke
X/Y/Z Accuracy:			
Linear Positioning Accuracy	+/- .005"	+/- .005"	+/- .005"
Linear Positioning Repeatability	+/- .003"	+/- .003"	+/- .003"
Maximum Rapid Traverse Speed	1200 IPM	1200 IPM	1200 IPM
Maximum Contour Speed	600 IPM	600 IPM	600 IPM
Acceleration/Deceleration	.05 g	.05 g	.05 g
C/B Accuracy:	C-Axis	B-Axis	
Rotary Axis Travel	±360°	±90°	
Rotary Positioning Accuracy	±.50°	±.50°	
Rotary Positioning Repeatability	±.25°	±.25°	
Maximum Programmable Speed	$90^\circ/s^2$	$90^\circ/s^2$	
Acceleration/Deceleration	$20^\circ/s^2$	$20^\circ/s^2$	

Table C.1. The AF-Series 5-Axis Three Dimensional Shapecutting CNC Machine specification (Flow Robotics Company, 1999).

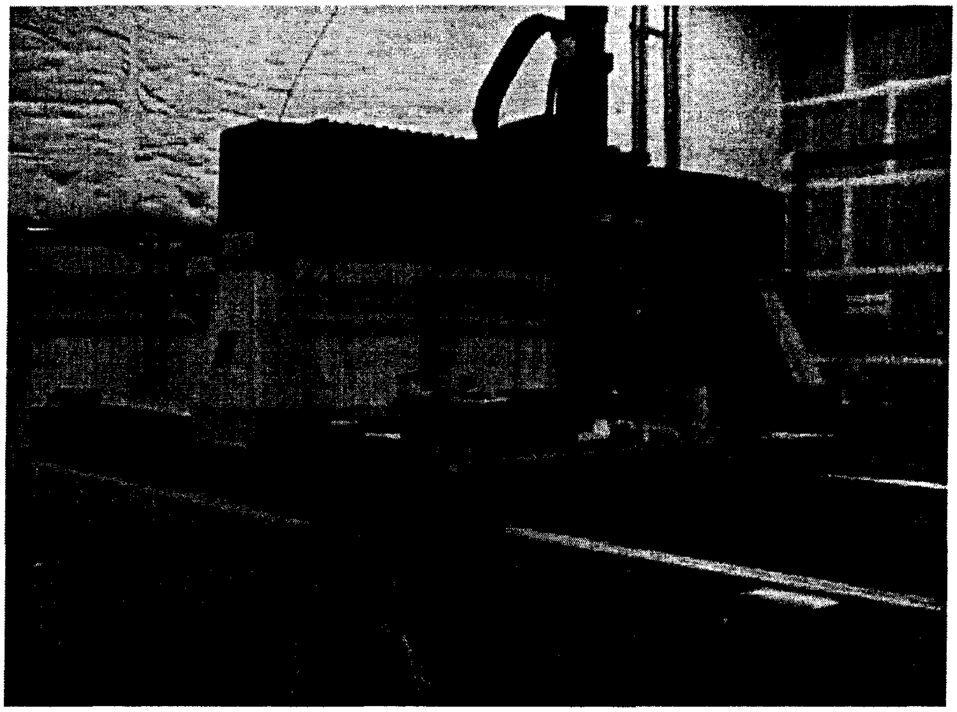


Figure C.1. Flow Robotics AF-Series 5-Axis Three Dimensional Shapecutting Machine.

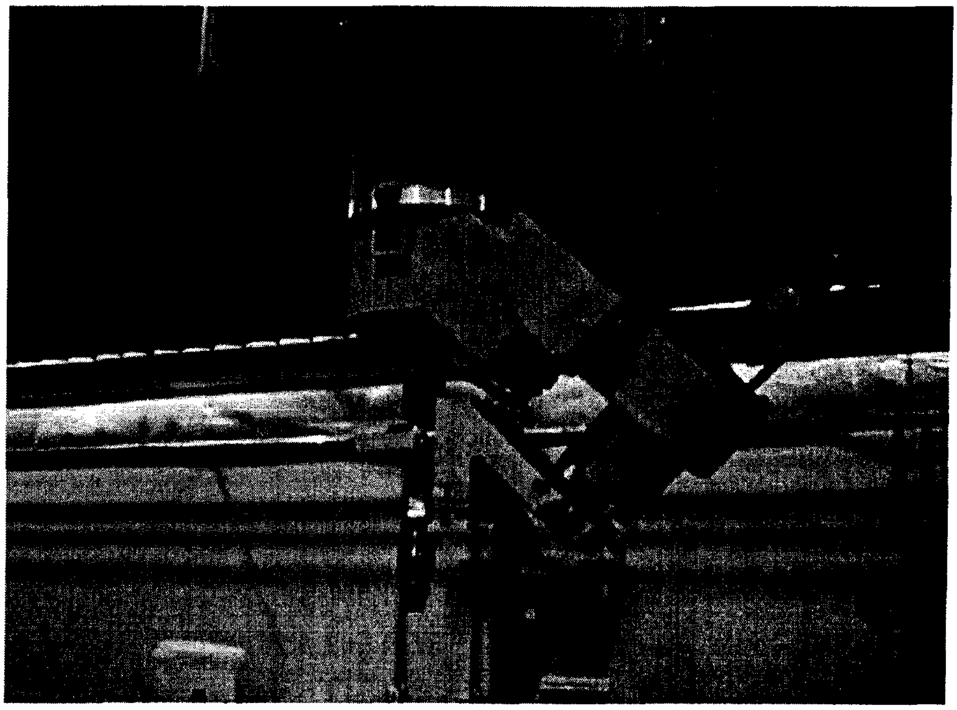


Figure C.2. The machine's rotational axes (B and C).

C.1. Standard Waterjet Shapecutter G-codes

CODE	PURPOSE
G00	Rapid point to point motion
G01	Linear interpolation
G02	Circular/helical, clockwise motion
G03	Circular/helical, counterclockwise motion
G04	Dwell time in seconds
G09	Exact stop
G14	Scaling (enable)
G14.1	Scaling (disable)
G15	Polar coordinate cancel
G16	Polar coordinate activate
G17	X, y plane selection
G18	X, z plane selection
G19	Y, z plane selection
G20	Inch mode active
G21	Metric mode active
G22	Programmable zone on
G23	Programmable zone off
G40	Reset cutter compensation
G41	Left cutter compensation
G42	Right cutter compensation
G52	Offset coordinate zero point
G53	Motion in machine coordinate system
G54	Preset work coordinate system 1
G55	Preset work coordinate system 2
G61	Exact stop mode
G64	Cutting mode
G90	Absolute dimensions
G91	Incremental dimensions
G92	Coordinate system offset
G92.1	Cancel coordinate system offset
G93	Inverse time feed mode
G94	Feed-per-minute mode

C.2. CNC M Codes

CODE	DESCRIPTION
M00	Program stop
M0l	Conditional program stop
M02	Program end
M30	Program stop/tape rewind
M98	Sub-program call
M99	Sub-program end and return

C.3. I/O M Codes

CODE	DESCRIPTION
M08	Dispense on (pressurized water on at nozzle)
M09	Dispense off (pressurized water off at nozzle)
M10	Abrasive on at nozzle
M11	Abrasive off at nozzle

APPENDIX D: CIRCULAR INTERPOLATION MAPPING

Circular interpolation of Allen-Bradley CNC controller is defined in five different ways:

- Case 1: Centre and end point parameters are known,
- Case 2: Radius and end point parameters are known,
- Case 3: Arc angle and centre point or end point parameters are known,
- Case 4: Polar coordinates are known,
- Case 5: CIP - start, intermediate and end point parameters are known.

It is because robot controllers can calculate a circular trajectory only if the start, intermediate, and end points are known, and because Workspace motion planner has been designed according to Realistic Robot Simulation specification (therefore not taking into account CNC controllers) the following mathematical transformations were performed to make CNC to robot circular motion mapping possible.

Case 1 – Centre and End Point

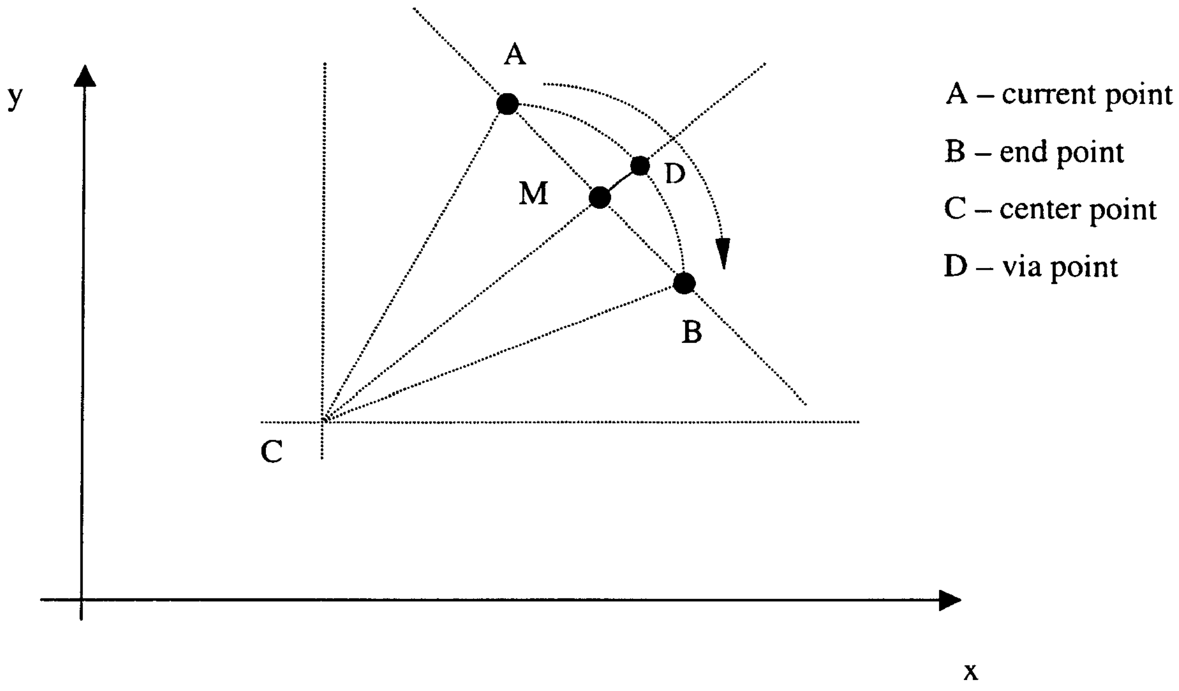


Figure D.1. Significant points used in calculation when centre and end point parameters are known.

Following calculation steps have been performed:

1 – Find the medium point M:

$$x_M = \frac{x_A + x_B}{2}$$

$$y_M = \frac{y_A + y_B}{2}$$

2 – Find the circle radius:

$$R = \sqrt{(x_A - x_C)^2 + (y_A - y_C)^2}$$

3 – Form the analytical equation of a circle, and the line that contains points M and C:

$$y - y_C = \frac{y_M - y_C}{x_M - x_C}(x - x_C)$$

$$(x - x_C)^2 - (y - y_C)^2 = R^2$$

4 – Introduce a new variable C$_1$:

$$C_1 = \frac{y_M - y_C}{x_M - x_C}$$

$$y - y_C = C_1(x - x_C)$$

5 – Put the above expression in the circle equation and solve it for x:

$$x = x_C \pm \sqrt{\frac{R^2}{1 + C_1^2}}$$

The calculated x coordinate belongs to the intersection point, which position needs to be determined. There are two solutions for x. Depending on the type of circular interpolation (clockwise or counter-clockwise), a plus sign or a minus sign in the previous equation will be used respectively.

One solution is located on the arc between points A and B, therefore:

$$x_A \le x_D \le x_B$$

$$y_A \le y_D \le y_B$$

6 –When a start point, an end point and a via point are known, the motion planner can accept those parameters and calculate a circular trajectory.

Special Cases:

There are two special cases:

- Points A and B are on the vertical line. In that case: $x_D = x_C + R$; $y_D = y_C$.
- Points A and B are on the horizontal line. Therefore: $x_D = x_C$; $y_D = y_C + R$.

Case 2 – Radius and End Point

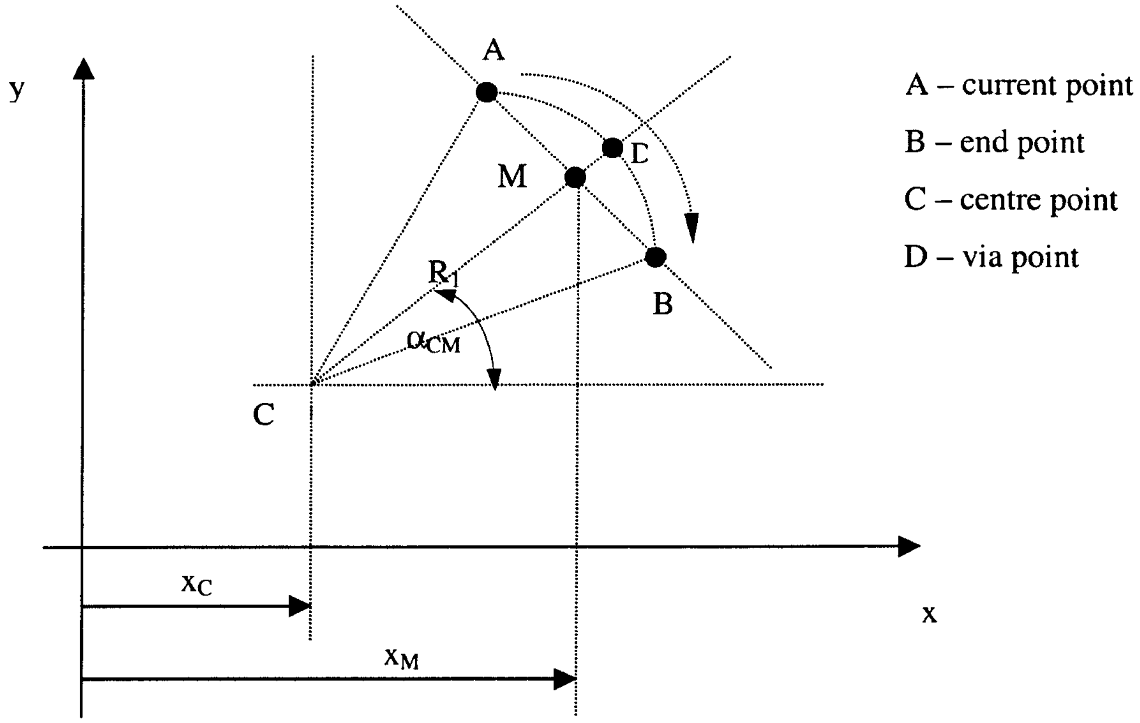


Figure D.2. Significant parameters used in calculation when circle radius and end point are known.

The beginning calculation steps are the same as for the case 1.

1 – Find the coordinates of point M.

2 – Find the length of line MB.

3 – Calculate the position of point C:

$$\begin{aligned} x_M &= x_C + R_1 \cos\alpha_{CM} \\ y_M &= y_C + R_1 \sin\alpha_{CM} \end{aligned} \quad \Rightarrow \quad \begin{aligned} x_C &= x_M - R_1 \cos\alpha_{CM} \\ y_C &= y_M - R_1 \sin\alpha_{CM} \end{aligned}$$

Plus and minus signs used in previous two equations are applicable only to the case presented in Figure 2. The problem is that there is one more circle that can be created through points A and B, and its centre is located opposite of point C with respect to line that goes through points A and B. In that case signs in the above equations will be changed.

4 – Calculate the length of R_1:

$$R_1 = \sqrt{R^2 - \overline{MB}^2}$$

5 – Calculate α_{CM}:

$$tg\alpha_{CM} = C_2 = -\frac{1}{C_1} \Rightarrow \alpha_{CM} = arctg(-\frac{1}{C_1})$$

$C_1 = tg\, \alpha$ (refer to Case 1)

6 – Therefore, after performing the described calculations the following parameters are known: start point, end point, and centre point, and that is the case 1.

<u>Special Cases:</u>

Since the function *arctg* was used, there is a possibility of error generation for specific angle values (0 and 90 degrees). Those cases have to be programmed separately:

- Line AB is horizontal (α_{CM}= 0 degrees)

$$x_C = x_M$$
$$y_C = y_M - R_1$$

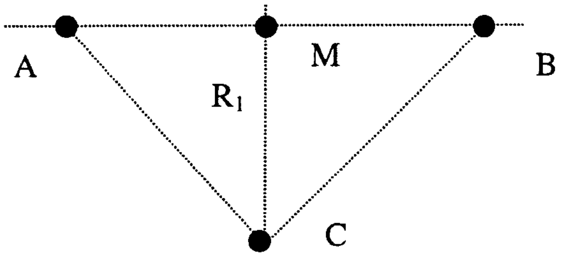


Figure D.3. Special case 1 (α_{CM}= 0 degrees), when circle radius and end point are known.

- Line AB is vertical (α_{CM} = 90 degrees)

$$x_C = x_M - R_1$$
$$y_C = y_M$$

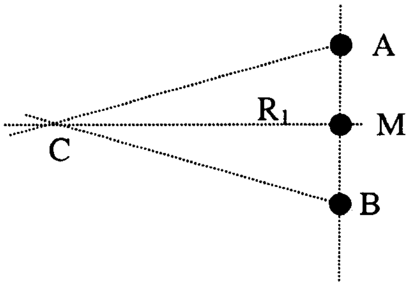


Figure D.4. Special case 1 (α_{CM}= 0 degrees), when circle radius and end point are known.

Case 3 – Centre Point or End Point and Angle

There are two possible sub-cases within this case:

Sub-case 1 – Known parameters: Angle and Centre Point

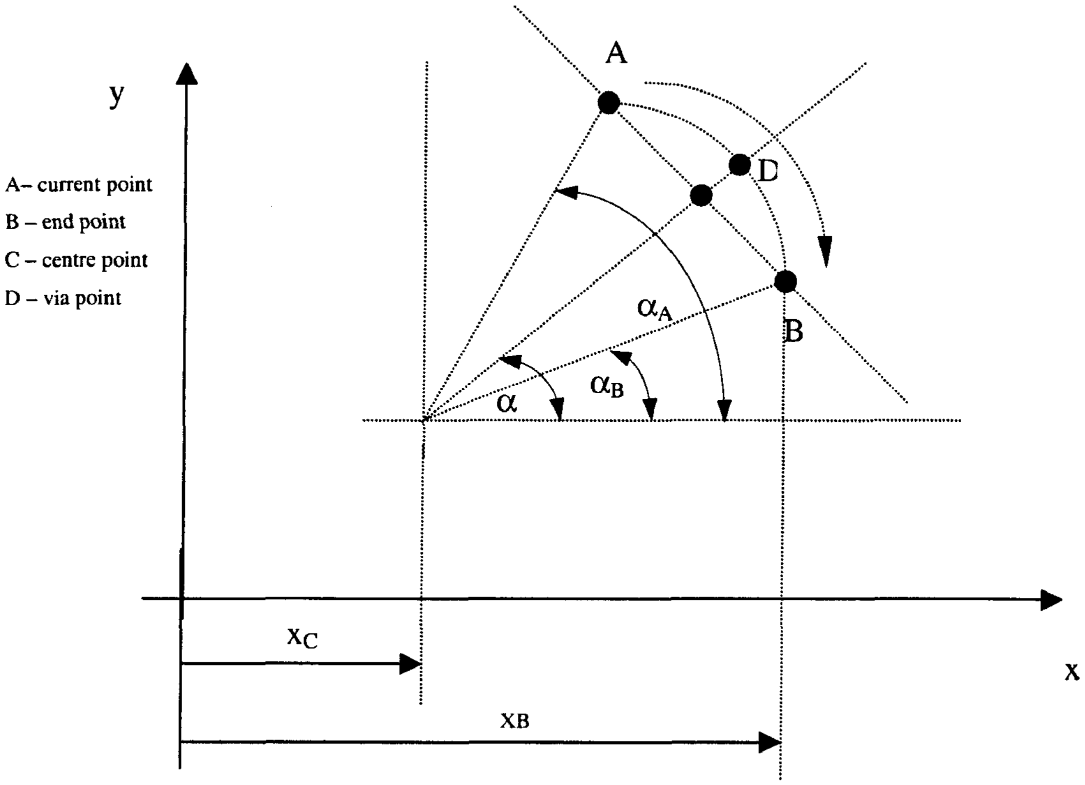


Figure D.5. Significant parameters used in calculation when arc centre point and arc central angle are known.

Following calculation steps have been performed:

1 – Calculate the circle radius using points A and C

2 – If the direction of revolution is clockwise then:

$$\alpha_A = \alpha + \alpha_B \Rightarrow \alpha_B = \alpha_A - \alpha$$

- If the direction of revolution is counter-clockwise then:

$$\alpha_B = \alpha + \alpha_A$$

3 – Calculate the target point coordinates:

$$x_B = x_C + R\cos\alpha_B$$
$$y_B = y_C + R\sin\alpha_B$$

4 – after performing the calculations shown in previous steps, the following parameters are known: a start point, an end point, and a centre point – which is the same as case 1 described above.

Sub-case 2 – Known parameters: start point, end point and angle

Following calculation steps have been performed:

1 – Find the medium point M between the points A and B

2 – Find the radius:

$$\sin\frac{\alpha}{2}=\frac{\overline{A}\overline{M}}{R}\Rightarrow R=\frac{AM}{\sin\left(\alpha/2\right)}$$

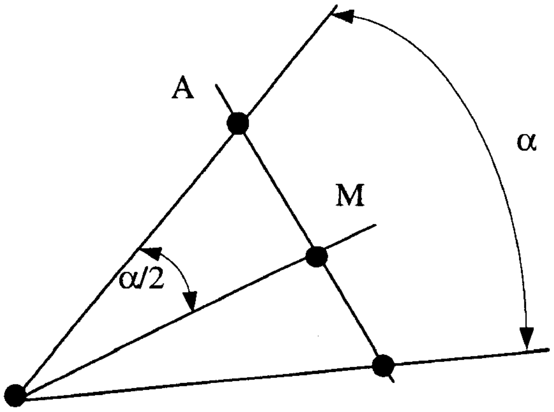


Figure D.6. Significant parameters used in calculation when start point, end point and the central angle are known.

In order to find the distance between points A and M, refer to case 2.

Case 4 – Polar Coordinates

The next parameters are known:

- polar angle,
- polar radius, and
- current point.

Pole is located in the circle centre point. This case is exactly the same as the case 3.1.

Case 5 – CIP – Arc or Circle through Intermediate Point

This is the standard motion planner case, therefore it will not be discussed.

APPENDIX E: G-CODE LANGUAGE GRAMMAR SCRIPT

```
#define TOKEN  int

#define VALUE  int

/* Redefining Error Macros */

#define BAD_ERROR(x) fprintf(stderr,"%s(%d) : failed parse:\
 probable        error        near        ..        <%s>        ..\nSkipping
...\n",p_infile,yylineno,yytext);\
 if(numerrs++ > 100) { fprintf(stderr, "\nToo many errors (101) ...
exiting"); \
 exit(numerrs);}while(yytchar)     {if(yytchar     <     1)     exit(-1);
get1token();}/*get1token();*/\
 yylloc=NULL;/*getchar()*/;

#define ZER_ERROR(x) fprintf(stderr,"%s(%d) : incomplete parse:\
 possible        error        near        ..        <%s>        ..\nSkipping
...\n",p_infile,yylineno,yytext);\
 if(numerrs++ > 100) { fprintf(stderr, "\nToo many errors (101) ...
exiting"); \
 exit(numerrs);}while(yytchar)     {if(yytchar     <     1)     exit(-1);
get1token();}/*get1token();*/\
 yylloc=NULL;/*getchar()*/;

#define BTK_ERROR(x) if (!p_entry){\
 if(numerrs++ > 100) { fprintf(stderr, "\nToo many errors (101) ...
exiting"); \
 exit(numerrs);}while(yytchar)     {if(yytchar     <     1)     exit(-1);
get1token();}/*get1token();*/\
 yylloc=NULL;\
 fprintf(stderr,"(line %d) error: parse backtracked across cut from
point near .. <%c> ..\n",yylineno,*(char*)maxp);}\
yylloc=NULL;\
longjmp(jmpb,1); /* should be to the ! we're backtracking across */\
/*getchar()*/;

#define END {DestroyASTs(); printf("\nTranslation Complete\n\n");}

#define TO_INT (int)

#define TO_CP (int *)

#include "Gcodes.y.h"
#include "ccx.h"
#include "AST Stuff.h"
#include <stdio.h>
#include <ctype.h>

char * yytemp;
extern char * yytext;
extern FILE * yyin;
FILE * output_file;
int nNumberOfInitFuncs = 0;
int nNumberOfInits = 0;
```

```
#define VAR_TABLE_SIZE 1024


int numerrs = 0;
int justOne = 0;


/*Numeric Values*/


@ num_value       =  point\x integer\y
            {@ ukey(TO_CP $x, ($y - $x)) @}
@                         |  first_digit\x [integer] point integer\y
            {@ ukey(TO_CP $x, ($y - $x)) @}
@                         |  first_digit\x [integer] point\y
            {@ ukey(TO_CP $x, ($y - $x)) @}
@                         |  first_digit\x integer\y
                  {@ ukey(TO_CP $x, ($y - $x)) @}
@                         |  first_digit\x
                  {@ ukey(TO_CP $x, 1) @}
@                     |  msign\x point integer\y
            {@ ukey(TO_CP $x, ($y - $x)) @}
@                         |  msign\x first_digit [integer] point integer\y
      {@ ukey(TO_CP $x, ($y - $x)) @}
@                         |  msign\x first_digit [integer] point\y
      {@ ukey(TO_CP $x, ($y - $x)) @}
@                         |  msign\x first_digit integer\y
            {@ ukey(TO_CP $x, ($y - $x)) @}
@                         |  msign\x first_digit\y
                  {@ ukey(TO_CP $x, ($y - $x)) @}

@ msign                 = <'-'> {@ TO_INT pstr @}

@ point                 = <'.'> {@ TO_INT pstr @}

@ first_digit     = (myisdigit) {@ TO_INT pstr @}

@ integer         = (myisdigit)*  {@ TO_INT pstr @}


/*Comments*/

@ comment         = behold\x           {: fprintf(output_file, "\n\nRem
%s", getstring($x)); :}

@ behold          =  letter('(')\x  <SPACE>*  text  letter(')')\y    {@
ukey(TO_CP $x, ($y - $x)) @}

@ text                  = contens+

@ contens         = word separator word

@ separator       = <SPACE>

@ word                  = (myisprint)* {@ TO_INT pstr @}

@ letter(n)       = <n> {@ (int) pstr @}


/*Program Structure*/

@ program_flow    = [<PERC>]                          <SPACE>*
@                         $*                                <SPACE>*
@                          [comment] [<SEMIC>]        <SPACE>*
@                          $*                               <SPACE>*
@                        program_name                       <SPACE>*
@                          $*                               <SPACE>*
@                          [comment] [<SEMIC>]        <SPACE>*
@                          $*                               <SPACE>*
@                          codes
      <SPACE>*
@                          $*                               <SPACE>*
@                          [comment] [<SEMIC>]        <SPACE>*
@                          $*                               <SPACE>*
@                          mainend                          <SPACE>*
@                          $*                               <SPACE>*
@                          subprograms* [<SEMIC>]     <SPACE>*
@                          <EOF>


@ program_name     = <'O'> [num_value]\p [<SEMIC>] {: static int counter
= 0;
@
if(counter == 0) {
@
      fprintf(output_file, "\n\nPublic Sub ReadyToGo ()");
@                                                                     }
@                                                                     else
{
@
      fprintf(output_file,     "\n\nPublic     Sub     Program%s     ()",
getstring($p));
@                                                                     }
@
      counter++;
@
      justOne =0;
@                                                              :}

@ mainend          = [<PERC>]  {:
@                                          if(justOne == 0)
@                                                {fprintf(output_file,
"\n\nEnd Sub");
@                                                      justOne++;
@                                                } :}

@ subprograms      = [comment] [<SEMIC>] <SPACE>*
@                          $*                   <SPACE>*
@                        program_name               <SPACE>*
@                          $*                       <SPACE>*
@                          [comment] [<SEMIC>] <SPACE>*
@                          $*                       <SPACE>*
@                          codes                          <SPACE>*
@                          $*                       <SPACE>*
@                          [comment] [<SEMIC>] <SPACE>*
@                          $*                       <SPACE>*
@                          gobacktomain             <SPACE>*
@                          $*                       <SPACE>*


@ gobacktomain     = <M99> [<SEMIC>]   {: fprintf(output_file, "\n\nEnd
Sub");  :}

@ codes            =  sentence+ $*

@ sentence        = [ <SLALINE> ] {$ | n_word | g_word | m_word | t_word |
s_word | h_word | d_word | setOfArgs | comment}+ [<SEMIC>] <SPACE>*
```

```
@ setOfArgs       =  {[argumentx]  [argumenty]  [argumentz]  [argumentb]
[argumentc] [argumentf] [argumenti] [argumentj] [argumentk] [argumentr]}
theRest* $
@                                  {:             fprintf(output_file,
"\n\nPerformMotion"); :}

@ theRest         = g_word | m_word | t_word | d_word | h_word

@ n_word          = <'N'> num_value  <SPACE>*

@ t_word          = <'T'> num_value  <SPACE>*

@ s_word          = <'S'> num_value  <SPACE>*

@ h_word          = <'H'> num_value  <SPACE>*

@ d_word          = <'D'> num_value  <SPACE>*

@ g_word          =        gtempty
@                        | gt00
@                        | gt01
@                        | gt02
@                        | gt03
@                        | gt04
@                        | gt09
@                        | gt1012
@                        | gt14
@                        | gt141
@                        | gt15
@                        | gt16
@                        | gt17
@                        | gt18
@                        | gt19
@                        | gt20
@                        | gt21
@                        | gt22
@                        | gt23
@                        | gt27
@                        | gt28
@                        | gt29
@                        | gt30
@                        | gt52
@                        | gt53
@                        | gt54
@                        | gt55
@                        | gt56
@                        | gt57
@                        | gt58
@                        | gt59
@                        | gt591
@                        | gt592
@                        | gt593
@                        | gt61
@                        | gt90
@                        | gt91
@                        | gt92
@                        | gt921
@                        | gt93
@                        | gt94
@                        | gt50
@                        | gt74
@                        | gt95
@                        | gt31
@                                  | gt100
@                                  | gt65
@                                  | gt81
@                                  | gt76
@                                  | gt41
@                                  | gt43
@                                  | gt431
@                                  | gt101
@                                  | gt103
@                                  | gt741
@                                  | gt13
@                                  | gt102
@                                  | gt161
@                                  | gt24
@                                  | gt25
@                                  | gt26
@                                  | gt37
@                                  | gt38
@                                  | gt381
@                                  | gt46
@                                  | gt68
@                                  | gt73
@                                  | gt83
@                                  | gt8811
@                                  | gt8812
@                                  | gt8813
@                                  | gt8814
@                                  | gt8815
@                                  | gt8821
@                                  | gt8822
@                                  | gt8823
@                                  | gt8824
@                                  | gt8831
@                                  | gt8832
@                                  | gt8841
@                                  | gt8842
@                                  | gt885
@                                  | gt886
@                                  | gt891
@                                  | gt892

@ gtempty          =       <G05>   <SPACE>*
@                                  | <G05_1>  <SPACE>*
@                                  | <G05_2>  <SPACE>*
@                                  | <G05_3>  <SPACE>*
@                                  | <G05_4>  <SPACE>*
@                                  | <G11>    <SPACE>*
@                                  | <G12_1>  <SPACE>*
@                                  | <G12_2>  <SPACE>*
@                                  | <G12_3>  <SPACE>*
@                                  | <G36>    <SPACE>*
@                                  | <G36_1>  <SPACE>*
@                                  | <G39>    <SPACE>*
@                                  | <G39_1>  <SPACE>*
@                                | <G40>    <SPACE>*
@                                  | <G45>    <SPACE>*
@                                  | <G46_1>  <SPACE>*
@                                  | <G49>    <SPACE>*
@                                  | <G62>    <SPACE>*
@                                  | <G63>    <SPACE>*
@                                  | <G64>    <SPACE>*
@                                  | <G67>    <SPACE>*
@                                  | <G69>    <SPACE>*
@                                  | <G80>    <SPACE>*
@                              | <G92_2>  <SPACE>*
@                              | <G98>    <SPACE>*
@                              | <G99>    <SPACE>*


@ gt00                     = <G00>    <SPACE>* rapid_axeseq

@ rapid_axeseq =           [argumentx]
@                                  [argumenty]
@                                  [argumentz]
@                                  [argumentb]
@                                  [argumentc]
@                                  [argumentf]
@                          {:    fprintf(output_file, "\n\nActiveMotionType
= \"Rapid\"");
@                                fprintf(output_file,   "\nPerformMotion");
:}

@ argumentx        =    <'X'>  num_value\x    <SPACE>*  {@   $x   @}     {:
fprintf(output_file, "\n\nDefCoord.x = %s", getstring($x)); :}

@ argumenty        =    <'Y'>  num_value\y    <SPACE>*  {@   $y   @}     {:
fprintf(output_file, "\n\nDefCoord.y = %s", getstring($y)); :}

@ argumentz        =    <'Z'>  num_value\z    <SPACE>*  {@   $z   @}     {:
fprintf(output_file, "\n\nDefCoord.z = %s", getstring($z)); :}

@ argumentb        =    <'B'>  num_value\b    <SPACE>*  {@   $b   @}     {:
fprintf(output_file, "\n\nDefCoord.b = %s", getstring($b)); :}

@ argumentc        =    <'C'>  num_value\c    <SPACE>*  {@   $c   @}     {:
fprintf(output_file, "\n\nDefCoord.c = %s", getstring($c)); :}

@ argumentf        =    <'F'>  num_value\f    <SPACE>*  {@   $f   @}     {:
fprintf(output_file, "\n\nFeedrate = %s", getstring($f)); :}

@ argumenti        =    <'I'>  num_value\i    <SPACE>*  {@   $i   @}     {:
fprintf(output_file, "\n\nCircParam.i = %s", getstring($i)); :}

@ argumentj        =    <'J'>  num_value\j    <SPACE>*  {@   $j   @}     {:
fprintf(output_file, "\n\nCircParam.j = %s", getstring($j)); :}

@ argumentk        =    <'K'>  num_value\k    <SPACE>*  {@   $k   @}     {:
fprintf(output_file, "\n\nCircParam.k = %s", getstring($k)); :}

@ argumentr        =    <'R'>  num_value\r    <SPACE>*  {@   $r   @}     {:
fprintf(output_file, "\n\nCircParam.r = %s", getstring($r)); :}


@ gt01                     = <G01>  <SPACE>* linear_axeseq

@ linear_axeseq =                  [argumentx]
@                                       [argumenty]
@                                       [argumentz]
@                                       [argumentb]
@                                       [argumentc]
@                                       [argumentf]
@                          {:    fprintf(output_file, "\n\nActiveMotionType
= \"Linear\"");
@                                fprintf(output_file,   "\nPerformMotion");
:}

@ gt02                     =        <G02> <SPACE>* circw_axeseq
@ circw_axeseq  =         [argumentx]
@                                        [argumenty]
@                                        [argumentz]
@                                        [argumentr]
@                                        [argumenti]
@                                        [argumentj]
@                                        [argumentk]
@                                        [argumentb]
@                                        [argumentc]
@                                        [argumentf]
@                         {:    fprintf(output_file, "\n\nActiveMotionType
= \"Circular CW\"");
@                               fprintf(output_file,   "\nPerformMotion");
:}


@ gt03                    =        <G03> <SPACE>* circcw_axeseq

@ circcw_axeseq   =             [argumentx]
@                                        [argumenty]
@                                        [argumentz]
@                                        [argumentr]
@                                        [argumenti]
@                                        [argumentj]
@                                        [argumentk]
@                                        [argumentb]
@                                        [argumentc]
@                                        [argumentf]
@                         {:    fprintf(output_file, "\n\nActiveMotionType
= \"Circular CCW\"");
@                               fprintf(output_file,   "\nPerformMotion");
:}

@ gt04                    = <G04>      <SPACE>* {pparam | xparam | uparam}


@ pparam            = <'P'> num_value\p   <SPACE>* {: fprintf(output_file,
"\n\nCNC.Delay 1000 * %s#", getstring($p));  :}

@ xparam            = <'X'> num_value\x   <SPACE>* {: fprintf(output_file,
"\n\nCNC.Delay 1000 * %s#", getstring($x));  :}

@ uparam            = <'U'> num_value\u   <SPACE>* {: fprintf(output_file,
"\n\nCNC.Delay 1000 * %s#", getstring($u));  :}


@ gt09                    = <G09>  <SPACE>*      {:  fprintf(output_file,
"\n\nCNC.Delay 1");
@
       fprintf(output_file, "\nCNC.RRS.StopMotion"); :}

@ gt1012            =  <G10L2>     <SPACE>*  <'P'>  num_value\p    <SPACE>*
originx\x originy\y originz\z
@                                                    {:         if  ($p
== 1) fprintf(output_file, "\n\nTableCS = \"G54\"");
@                                                         else  if  ($p
== 2) fprintf(output_file, "\n\nTableCS = \"G55\"");
@                                                         else  if  ($p
== 3) fprintf(output_file, "\n\nTableCS = \"G56\"");
@                                                         else  if  ($p
== 4) fprintf(output_file, "\n\nTableCS = \"G57\"");
@                                                         else  if  ($p
== 5) fprintf(output_file, "\n\nTableCS = \"G58\"");
@                                                                  else  if  ($p
== 6) fprintf(output_file, "\n\nTableCS = \"G59\"");
@                                                                  else  if  ($p
== 7) fprintf(output_file, "\n\nTableCS = \"G59.1\"");
@                                                                  else  if  ($p
== 8) fprintf(output_file, "\n\nTableCS = \"G59.2\"");
@                                                                  else  if  ($p
== 9) fprintf(output_file, "\n\nTableCS = \"G59.3\"");
@
      fprintf(output_file, "\n\nIsItActive %s, %s, %s", getstring($x),
getstring($y), getstring($z));
@                                                           :}

@ originx         = <'X'> num_value\x  <SPACE>* {@ $x @}

@ originy         = <'Y'> num_value\y  <SPACE>* {@ $y @}

@ originz         = <'Z'> num_value\z  <SPACE>* {@ $z @}


@ gt14                  =   <G14>     <SPACE>*   {:   fprintf(output_file,
"\n\nScaleValue = 1"); :}

@ gt141                 = <G14_1>  <SPACE>* scalevalue

@ scalevalue      = <'P'> num_value\p  <SPACE>* {: fprintf(output_file,
"\n\nScaleValue = %s", getstring($p));  :}

@ gt15                  =   <G15>     <SPACE>*   {:   fprintf(output_file,
"\n\nIsPolar = False");
@
fprintf(output_file, "\nResetPolarAngles");:}

@ gt16                  =       {<G16> <SPACE>* argumentx argumenty} {:
      fprintf(output_file, "\n\nIsPolar = True\nPerformMotion"); :}
@                               | {<G16> <SPACE>* argumentz argumentx} {:
      fprintf(output_file, "\n\nIsPolar = True\nPerformMotion"); :}
@                               | {<G16> <SPACE>* argumenty argumentz} {:
      fprintf(output_file, "\n\nIsPolar = True\nPerformMotion"); :}
@                               {[argumentb]} {[argumentc]} {[argumentf]}


@ gt17                  =   <G17>     <SPACE>*   {:   fprintf(output_file,
"\n\nPlane = 17"); :}

@ gt18                  =   <G18>     <SPACE>*   {:   fprintf(output_file,
"\n\nPlane = 18"); :}

@ gt19                  =   <G19>     <SPACE>*   {:   fprintf(output_file,
"\n\nPlane = 19"); :}

@ gt20                  =   <G20>     <SPACE>*   {:   fprintf(output_file,
"\n\nMSValue = 25.4"); :}

@ gt21                  =   <G21>     <SPACE>*   {:   fprintf(output_file,
"\nMSValue = 1"); :}

@ gt22                  =  <G22>       <SPACE>*  {:  fprintf(output_file,
"\n\nZoneOn = True"); :}
@                        |   <G22_1>   <SPACE>*   {:   fprintf(output_file,
"\n\nZoneOn = True"); :}
```

```
@ gt23                    =  <G23>       <SPACE>*  {:  fprintf(output_file,
"\n\nZoneOn = False"); :}
@                           |  <G23_1>  <SPACE>*  {:  fprintf(output_file,
"\n\nZoneOn = False"); :}

@ gt27                    = <G27>  <SPACE>*
@                                    [argumentx]
@                                      [argumenty]
@                         [argumentz]
@                                      [argumentb]
@                                      [argumentc]
@                                      [argumentf]
@                         {:    fprintf(output_file, "\n\nG27"); :}

@ gt28                    = <G28>  <SPACE>*
@                                    [argumentx]
@                                      [argumenty]
@                         [argumentz]
@                                      [argumentb]
@                                      [argumentc]
@                                      [argumentf]
@                         {:    fprintf(output_file, "\n\nG28"); :}



@ gt29                    = <G29>  <SPACE>*
@                                    [argumentx]
@                                      [argumenty]
@                         [argumentz]
@                                      [argumentb]
@                                      [argumentc]
@                                      [argumentf]
@                         {:    fprintf(output_file, "\n\nG29"); :}

@ gt30                    = <G30>  <SPACE>*
@                                      [argumentx]
@                                      [argumenty]
@                         [argumentz]
@                                      [argumentb]
@                                      [argumentc]
@                                      [argumentf]
@                         {:    fprintf(output_file, "\n\nG30"); :}

@ gt52                    = <G52>  <SPACE>*
@                                      [markx]\x
@                                      [marky]\y
@                                      [markz]\z
@                                      {:              fprintf(output_file,
"\n\nMoveOrigin  52,  %s,  %s,  %s",  getstring($x),  getstring($y),
getstring($z)); :}


@ gt53                    = <G53>  <SPACE>*             {:
fprintf(output_file, "\n\nActiveCS = \"G53\"");
@
fprintf(output_file, "\nIsWCS = False"); :}

@ gt54                    = <G54>  <SPACE>*        {:  fprintf(output_file,
"\n\nActiveCS = \"G54\"");
@
fprintf(output_file, "\nIsWCS = True"); :}

@ gt55                    = <G55>     <SPACE>*          {:
fprintf(output_file, "\n\nActiveCS = \"G55\"");
@
fprintf(output_file, "\nIsWCS = True"); :}

@ gt56                    = <G56>      <SPACE>*         {:
fprintf(output_file, "\n\nActiveCS = \"G56\"");
@
fprintf(output_file, "\nIsWCS = True"); :}

@ gt57                    = <G57>      <SPACE>*         {:
fprintf(output_file, "\n\nActiveCS = \"G57\"");
@
fprintf(output_file, "\nIsWCS = True"); :}

@ gt58                    = <G58>      <SPACE>*         {:
fprintf(output_file, "\n\nActiveCS = \"G58\"");
@
fprintf(output_file, "\nIsWCS = True"); :}

@ gt59                    = <G59>      <SPACE>*         {:
fprintf(output_file, "\n\nActiveCS = \"G59\"");
@
fprintf(output_file, "\nIsWCS = True"); :}

@ gt591                   = <G59_1>  <SPACE>*           {:
fprintf(output_file, "\n\nActiveCS = \"G59.1\"");
@
fprintf(output_file, "\nIsWCS = True"); :}

@ gt592                   = <G59_2>  <SPACE>*           {:
fprintf(output_file, "\n\nActiveCS = \"G59.2\"");
@
fprintf(output_file, "\nIsWCS = True"); :}

@ gt593                   = <G59_3>  <SPACE>*           {:
fprintf(output_file, "\n\nActiveCS = \"G59.3\"");
@
fprintf(output_file, "\nIsWCS = True"); :}

@ gt61                  = <G61>  <SPACE>*          {:  fprintf(output_file,
"\n\nCNC.Delay 1");
@
      fprintf(output_file, "\nCNC.RRS.StopMotion"); :}


@ gt90                    = <G90>      <SPACE>*         {:
fprintf(output_file, "\n\nIsRelative = False");  :}


@ gt91                    = <G91>      <SPACE>*         {:
fprintf(output_file, "\n\nIsRelative = True"); :}


@ gt92                    = <G92>  <SPACE>*
@                                     [markx]\x
@                                     [marky]\y
@                                     [markz]\z
@                                     {:              fprintf(output_file,
"\n\nMoveOrigin   92,   %s,   %s,   %s",  getstring($x),   getstring($y),
getstring($z)); :}

@ markx             = <'X'> num_value\x  <SPACE>* {@ $x @}

@ marky             = <'Y'> num_value\y  <SPACE>* {@ $y @}
@ markz           = <'Z'> num_value\z  <SPACE>* {@ $z @}

@ gt921           =   <G92_1>      <SPACE>*   {:   fprintf(output_file,
"\n\nMoveOrigin 921, 0#, 0#, 0#"); :}

@ gt93                 = <G93>   <SPACE>* argumentf   <SPACE>*     {:
fprintf(output_file, "\n\nIsInverseTime = True"); :}

@ gt94                 = <G94>   <SPACE>* [argumentf]   <SPACE>* {:
fprintf(output_file, "\n\nIsInverseTime = False"); :}
```

APPENDIX F: C FUNCTION LIBRARY FOR PARSER CUSTOMIZATION

```
*****************************************************************
* @doc
* @module AST Stuff.c |
*
*
* Copyright (c) 1999 Robot Simulations Ltd.
*
* Filename: AST Stuff.c
* Created:  09/02/99
* Author:   Aleksandar Boskovic
*
* Compiler: Microsoft Visual C++ v6.0
* OS:       Win32
*
* Version:  v1.0
*
* Description:    Functions to help create Robot Language Translators
*
*****************************************************************/

#include "AST Stuff.h"


/*****************************************************************
* @func
* Function: newkey
*
* Arguments:
*     char *x,
*     long n
*
*
* Returns:  static int
*
* Date:           09/02/99
* Author:   Aleksandar Boskovic
*
* Description:    Adds a new string constant to the list
*
*****************************************************************/
static int newkey(char *x, long n)
/* put a new string in the stash and give it a key */
/* negative returns are errors */
{
    char *y;
    NODE *z;
    /* the minimum position is 1 */

    if (wordcount >= MAXWORDS - 1)
      {
            fprintf(stderr, "\nString list too short...Exiting");
            exit(-1);
      }
    y = malloc((unsigned)n + 1);
    if (!y)  /* out of memory */
        return -1;
    z = (NODE *)malloc(sizeof(NODE));
    if (!z)  /* out of memory */
        return -1;
    strncpy(y,x,(unsigned int)n);
    y[n] = 0;
    z->word = y;
    nodelist[++wordcount] = z;  /* store the pointer for comparisons */
    return wordcount;
}


/*****************************************************************
* @func
* Function: lookup
*
* Arguments:
*     char *x,
*     long n
*
*
* Returns:  static int
*
* Date:           09/02/99
* Author:   Aleksandar Boskovic
*
* Description:    Checks to see if the string passed is already in
*     in the list, if so, it returns the unique identifier
*
*****************************************************************/
static int lookup(char *x, long n)
/* find a name in the list
 * if not there return 0, else return its index
 */
{
    int i, test = 0;
    char *w;

      for(i = wordcount; i ; i--)
      {
            if(!nodelist[i]) //avoid problems with null pointers
              return 0;            //shouldn't happen
            w = nodelist[i]->word;
            if(!strcmp(x,w))
            {     //They're identical
                  test = i;
                  break;
            }
      }
      return test;
}


/*****************************************************************
* @func
* Function: ukey
*
* Arguments:
*     int *x,
*     long n
*
*
* Returns:  int
*
* Date:           09/02/99
* Author:   Aleksandar Boskovic
```

```
*
* Description:    Checks to see if a string is in the list, if not
*     then it adds it to the end
*
***************************************************************/
int ukey(int *x, long n)
/* look up str in notified word list and return unique key.
 * Add new key if necessary and return that. If we can't, return -1.
 */
{
      static char TempArray [2048];

    int i = 0;
      x--;  //Always one letter off!
      n= (n + sizeof(int))/ sizeof(int);  //convert to character string

      do
      {
            if(i > n)         //convert to character string
                  break;
            *(TempArray + i) = *(x + i);
            i++;
      }
      while(*(x + i));
      *(TempArray + n) = 0;

      i = lookup(TempArray ,n);
    if (i > 0)                               /* found */
        return i;
    i = newkey(TempArray,n);                         /* new key */
    return i;
}


/***************************************************************
* @func
* Function: getstring
*
* Arguments:
*     int ukey
*
*
* Returns:  char *
*
* Date:           09/02/99
* Author:   Aleksandar Boskovic
*
* Description:    Returns the string stored in the specified position
(if any)
*
***************************************************************/
char * getstring(int ukey)
{
      char * psRval; //if nothing just add a space

      if((ukey > 0) && (ukey <= wordcount))
      {
            if(nodelist[ukey])
                  psRval = nodelist[ukey]->word;
            else
                  psRval = Empty;
      }
      else
            psRval = Empty;
      return psRval;
}

/*******************************************************************
* @func
* Function: myisalpha
*
* Arguments:
*     TOKEN test
*
*
* Returns:  BOOLEAN
*
* Date:           09/02/99
* Author:   Aleksandar Boskovic
*
* Description:    checks if the token is an alpha character
*
*******************************************************************/
BOOLEAN myisalpha(TOKEN test)
{
      return ((((test >= 'a') && (test <= 'z')) || ((test >= 'A') &&
(test <= 'Z'))) ? 1 : 0);
}

/*******************************************************************
* @func
* Function: myisdigit
*
* Arguments:
*     TOKEN test
*
*
* Returns:  BOOLEAN
*
* Date:           09/02/99
* Author:   Aleksandar Boskovic
*
* Description:    checks if the token is a digit
*
*******************************************************************/
BOOLEAN myisdigit(TOKEN test)
{
      return (((test >= '0') && (test <= '9')) ? 1 : 0);
}


/*******************************************************************
* @func
* Function: myisalnum
*
* Arguments:
*     TOKEN test
*
*
* Returns:  BOOLEAN
*
* Date:           09/02/99
* Author:   Aleksandar Boskovic
*
* Description:    checks if the token is a digit, alpha or '_'
*
*******************************************************************/
```

```
BOOLEAN myisalnum(TOKEN test)
{
      return ( myisdigit(test) || myisalpha(test) || (test == '_')) ? 1
: 0;
}

/*****************************************************************
* @func
* Function: myisprint
*
* Arguments:
*     TOKEN test
*
*
* Returns:  BOOLEAN
*
* Date:           09/02/99
* Author:   Aleksandar Boskovic
*
* Description:    checks if the token is a printable character
*
****************************************************************/
BOOLEAN myisprint(TOKEN test)
{
      return ((( isprint(test) || (test == ' ')) && ((test != '(') &&
(test != ')') && (test != '%'))) ? 1 : 0);
}
```

APPENDIX G: WORKSPACE SIMULATION LANGUAGE LIBRARY FILE

```
'Data Declaration

'User Type
Type JointSet
x As Double
y As Double
z As Double
b As Double
c As Double
End Type

Type PosAndOr
x As Double
y As Double
z As Double
a As Double
b As Double
c As Double
End Type

Type CircIntArg
r As Double
i As Double
j As Double
k As Double
End Type

Type Zone
Xmin As Double
Xmax As Double
Ymin As Double
Ymax As Double
Zmin As Double
Zmax As Double
End Type

Type Transformed
x As Double
y As Double
z As Double
End Type

Type WorkCoordSys
Gcode As String
Props As PosAndOr
End Type

'Constants
Public Const PI As Double = 3.14159265
Public Const MAXROTSPEED As Double = 5400#
Public Const RAPIDFEEDRATE As Double = 30480#


'Variables
```

```
Public IsPolar As Boolean
Public ZoneOn As Boolean
Public IsRelative As Boolean
Public IsWCS As Boolean
Public IsInverseTime As Boolean
Public Feedrate As Double
Public MSValue As Double
Public ScaleValue As Double
Public Radius As Double
Public CutterComp As Double
Public ToolLenght As Double
Public PolAng17 As Double
Public PolAng18 As Double
Public PolAng19 As Double
Public CuttingMode As Integer
Public GPCounter As Integer
Public Plane As Integer
Public MotionValue As Integer
Public ActiveMotionType As String
Public ActiveCS As String
Public TableCS As String
Public Configuration As String

'Type variables
Public NewZone As Zone
Public CircParam As CircIntArg
Public CenterRadius As Transformed
Public DefCoord As JointSet
Public PrevCoord As JointSet
Public NewPrimaryHome As JointSet
Public NewSecondaryHome As JointSet
Public MemoCoord As JointSet
Public MCSTarget As JointSet
Public NewTarget As JointSet
Public WorkCS As WorkCoordSys

'Objects
Public CNC As New Robot
Public JointPos As New RCSJointPos
Public CartPos As New RCSCartPos
Public NewFrame As New RCSFrame

Public Sub JointValues(tx As Double, ty As Double, tz As Double, tb As
Double, tc As Double)
  Dim AfterTrans As Transformed
  Dim x As Double, y As Double, z As Double, b As Double, c As Double

    tx = ToMM(tx)
    ty = ToMM(ty)
    tz = ToMM(tz)

If IsPolar Then

  If IsRelative Then

    If Plane = 17 Then
        PolAng17 = ty + PolAng17
        x = tx * Cdeg(PolAng17)
        y = tx * Sdeg(PolAng17)
        z = tz
        ElseIf Plane = 19 Then
        PolAng19 = tz + PolAng19
        x = tx
        y = ty * Cdeg(PolAng19)
```

```
         z = ty * Sdeg(PolAng19)
         ElseIf Plane = 18 Then
         PolAng18 = tx + PolAng18
         x = tz * Sdeg(PolAng18)
         y = ty
         z = tz * Cdeg(PolAng18)
     End If
   End If

   If Not IsRelative Then
     If Plane = 17 Then
         x = tx * Cdeg(ty)
         y = tx * Sdeg(ty)
         z = tz
     ElseIf Plane = 19 Then
         x = tx
         y = ty * Cdeg(tz)
         z = ty * Sdeg(tz)
     ElseIf Plane = 18 Then
         x = tz * Sdeg(tx)
         y = ty
         z = tz * Cdeg(tx)
     End If
 End If

Else
    x = tx
    y = ty
    z = tz
    b = tb
    c = tc
End If

   If IsRelative Then
   MCSTarget.x = x + MCSTarget.x
   MCSTarget.y = y + MCSTarget.y
   MCSTarget.z = z + MCSTarget.z
   MCSTarget.b = b + MCSTarget.b
   MCSTarget.c = c + MCSTarget.c

   Else
   MCSTarget.x = x
   MCSTarget.y = y
   MCSTarget.z = z
   MCSTarget.b = b
   MCSTarget.c = c

   End If

   NewTarget = MCSTarget

   If IsWCS Then
        If ActiveCS Like WorkCS.Gcode Then
        SetFrame WorkCS
        AfterTrans = TransformCoords(NewFrame, NewTarget.x, NewTarget.y,
NewTarget.z)
        End If
   NewTarget.x = AfterTrans.x
   NewTarget.y = AfterTrans.y
   NewTarget.z = AfterTrans.z

   End If

    JointPos.AxesFormat = 1
    JointPos.AxesFlags = 31

    JointPos.SetAxesValue 0, NewTarget.x
    JointPos.SetAxesValue 1, NewTarget.y
    JointPos.SetAxesValue 2, NewTarget.z
    JointPos.SetAxesValue 3, DegToRad(NewTarget.c)
    JointPos.SetAxesValue 4, DegToRad(NewTarget.b)


    CartPos.flag = 0

    If  ZoneOn  And  (NewTarget.x  <  NewZone.Xmax  And  NewTarget.x  >
NewZone.Xmin  And  NewTarget.y  <  NewZone.Ymax  And  NewTarget.y  >
NewZone.Ymin  And  NewTarget.z  <  NewZone.Zmax  And  NewTarget.z  >
NewZone.Zmin) Then
    CNC.RRS.StopMotion
    End If

End Sub

Public Sub DefineMotionType(mt As Integer)
MotionValue = mt
End Sub

Public Function ReturnMotionType() As Integer
ReturnMotionType = MotionValue
End Function

Public Sub SetDefaults()

IsWCS = False
Plane = 17
ScaleValue = 1
MSValue = 1
MotionValue = 2
Configuration = "RCSUndefinedConfiguration"
ActiveMotionType = "Rapid"
ActiveCS = "G53"
Feedrate = RAPIDFEEDRATE
ResetCircParam

End Sub

Public Sub JointHomeValues(i As Integer)

    JointPos.AxesFormat = 1
    JointPos.AxesFlags = 31

    If (i = 1) Then
        JointPos.SetAxesValue 0, NewPrimaryHome.x
        JointPos.SetAxesValue 1, NewPrimaryHome.y
        JointPos.SetAxesValue 2, NewPrimaryHome.z
        JointPos.SetAxesValue 3, DegToRad(NewPrimaryHome.c)
        JointPos.SetAxesValue 4, DegToRad(NewPrimaryHome.b)


    ElseIf (i = 2) Then
        JointPos.SetAxesValue 0, NewSecondaryHome.x
        JointPos.SetAxesValue 1, NewSecondaryHome.y
        JointPos.SetAxesValue 2, NewSecondaryHome.z
        JointPos.SetAxesValue 3, DegToRad(NewSecondaryHome.c)
        JointPos.SetAxesValue 4, DegToRad(NewSecondaryHome.b)
    ElseIf (i = 3) Then
        JointPos.SetAxesValue 0, MemoCoord.x
        JointPos.SetAxesValue 1, MemoCoord.y
        JointPos.SetAxesValue 2, MemoCoord.z
        JointPos.SetAxesValue 3, DegToRad(MemoCoord.c)
        JointPos.SetAxesValue 4, DegToRad(MemoCoord.b)

    End If

    CartPos.flag = 0

End Sub

Public Sub MemorizeCoordinates(x As Double, y As Double, z As Double, b
As Double, c As Double)
MemoCoord.x = x
MemoCoord.y = y
MemoCoord.z = z
MemoCoord.b = b
MemoCoord.c = c
End Sub

Public Function GetArcCenter(i As Double, j As Double, k As Double) As
RCSCartPos
Dim ArcCenter As New RCSCartPos
ArcCenter.flag = 3
ArcCenter.nx = 0#
ArcCenter.ny = 0#
ArcCenter.nz = 0#
ArcCenter.ax = 0#
ArcCenter.ay = 0#
ArcCenter.az = 0#
ArcCenter.ox = 0#
ArcCenter.oy = 0#
ArcCenter.oz = 0#
ArcCenter.px = CDbl(PrevCoord.x + i)
ArcCenter.py = CDbl(PrevCoord.y + j)
ArcCenter.pz = CDbl(PrevCoord.z + k)

Set GetArcCenter = ArcCenter

End Function

Public Sub RunTrack(RobotName As String)
Set CNC = ThisDocument.GetRobot(RobotName)
InsteadOfForm
SetDefaults
ReadyToGo
End Sub

Public Function TransformCoords(Frame As RCSFrame, oldX As Double, oldY
As Double, oldZ As Double) As Transformed
Dim MyTransformed As Transformed

MyTransformed.x = Frame.nx * oldX + Frame.ox * oldY + Frame.ax * oldZ +
Frame.px
MyTransformed.y = Frame.ny * oldX + Frame.oy * oldY + Frame.ay * oldZ +
Frame.py
MyTransformed.z = Frame.nz * oldX + Frame.oz * oldY + Frame.az * oldZ +
Frame.pz
TransformCoords = MyTransformed
End Function

Public Function Cdeg(angle As Double) As Double
```

```
Cdeg = Cos(angle * PI / 180)
End Function

Public Function Sdeg(angle As Double) As Double
Sdeg = Sin(angle * PI / 180)
End Function

Public Function ToMM(coord As Double) As Double
    ToMM = MSValue * ScaleValue * coord
End Function

Public Sub IsItActive(x As Double, y As Double, z As Double)
If TableCS Like WorkCS.Gcode Then
NewFrame.px = x
NewFrame.py = y
NewFrame.pz = z
End If
End Sub

Public Sub SetFeedrate(f As Double)
Dim CentralAngle As Double, LinDist As Double, TimeInMs As Double
If IsInverseTime Then
    If MotionValue = 2 Then
    TimeInMs = (60000 / f)
    CNC.RRS.SetMotionTime TimeInMs
    ElseIf MotionValue = 4 Then
    CentralAngle = CalcCentralAngle()
    CNC.RRS.SetMotionTime CentralAngle * 60 * 1000 / f
    End If
Else

    CNC.RRS.SetCartesianPositionSpeed f / 60
    CNC.RRS.SetCartesianOrientationSpeed 2, f / 60

End If

End Sub

Public Sub MoveOrigin(num As Integer, x As Double, y As Double, z As
Double)
If num = 92 Then
IsWCS = True
ActiveCS = "G54"
WorkCS.Props.x = PrevCoord.x - x
WorkCS.Props.y = PrevCoord.y - y
WorkCS.Props.z = PrevCoord.z - z
    ElseIf num = 52 Then
    IsWCS = True
    ActiveCS = "G54"
    WorkCS.Props.x = WorkCS.Props.x + x
    WorkCS.Props.y = WorkCS.Props.y + y
    WorkCS.Props.z = WorkCS.Props.z + z
ElseIf num = 921 Then
SetWorkCSParam
End If

End Sub

Public Function Calc2PointDist(x1 As Double, y1 As Double, z1 As Double,
x2 As Double, y2 As Double, z2 As Double) As Double
Dim Number As Double
Number = (x2 - x1) ^ 2 + (y2 - y1) ^ 2 + (z2 - z1) ^ 2
Calc2PointDist = Sqr(Number)
End Function
```

```
Public Function CosRule(a As Double, b As Double, c As Double) As Double
Dim cosalpha As Double
If a <> 0 And b <> 0 And c <> 0 Then
cosalpha = (b ^ 2 + c ^ 2 - a ^ 2) / (2 * b * c)
CosRule = Atn(-cosalpha / Sqr(-cosalpha * cosalpha + 1)) + 2 * Atn(1)
End If
End Function

Public Function CalcCenterPoint(i As Double, j As Double, k As Double)
As Transformed
CenterRadius.x = PrevCoord.x + i
CenterRadius.y = PrevCoord.y + j
CenterRadius.z = PrevCoord.z + k
CalcCenterPoint = CenterRadius
End Function

Public Function CalcCentralAngle() As Double
Dim adist As Double, angle As Double
    CalcCenterPoint CircParam.i, CircParam.j, CircParam.k
    adist = Calc2PointDist(PrevCoord.x, PrevCoord.y, PrevCoord.z,
DefCoord.x, DefCoord.y, DefCoord.z)
    Radius = Calc2PointDist(CenterRadius.x, CenterRadius.y,
CenterRadius.z, DefCoord.x, DefCoord.y, DefCoord.z)
    angle = CosRule(adist, Radius, Radius)
    CalcCentralAngle = angle
End Function

Public Sub SetFrame(Values As WorkCoordSys)

NewFrame.nx = Cdeg(Values.Props.c) * Cdeg(Values.Props.a) -
Sdeg(Values.Props.c) * Cdeg(Values.Props.b) * Sdeg(Values.Props.a)
NewFrame.ny = Sdeg(Values.Props.c) * Cdeg(Values.Props.a) +
Cdeg(Values.Props.c) * Cdeg(Values.Props.b) * Sdeg(Values.Props.a)
NewFrame.nz = Sdeg(Values.Props.b) + Sdeg(Values.Props.a)
NewFrame.ox = -Cdeg(Values.Props.c) * Sdeg(Values.Props.a) -
Sdeg(Values.Props.c) * Cdeg(Values.Props.b) * Cdeg(Values.Props.a)
NewFrame.oy = -Sdeg(Values.Props.c) * Sdeg(Values.Props.a) +
Cdeg(Values.Props.c) * Cdeg(Values.Props.b) * Cdeg(Values.Props.a)
NewFrame.oz = Sdeg(Values.Props.b) * Cdeg(Values.Props.a)
NewFrame.ax = Sdeg(Values.Props.c) * Sdeg(Values.Props.b)
NewFrame.ay = -Cdeg(Values.Props.c) * Sdeg(Values.Props.b)
NewFrame.az = Cdeg(Values.Props.b)
NewFrame.px = Values.Props.x
NewFrame.py = Values.Props.y
NewFrame.pz = Values.Props.z

End Sub

Public Sub PerformMotion()

If ActiveMotionType Like "Rapid" Then
CNC.RRS.SelectMotionType 2
DefineMotionType 2
JointValues DefCoord.x, DefCoord.y, DefCoord.z, DefCoord.b, DefCoord.c
CNC.RRS.SetCartesianPositionSpeed RAPIDFEEDRATE / 60
CNC.RRS.SetCartesianOrientationSpeed 2, MAXROTSPEED / 60
PrevCoord = NewTarget
GPCounter = GPCounter + 1
CNC.RRS.SetNextNamedTarget 0, 0, CartPos, JointPos, Configuration, 0,
"GP" & CStr(GPCounter)

    ElseIf ActiveMotionType Like "Linear" Then
    Dim CriticalAngle As Double
    CNC.RRS.SelectMotionType 2
    DefineMotionType 2
    SetFeedrate Feedrate
    JointValues   DefCoord.x,   DefCoord.y,   DefCoord.z,   DefCoord.b,
DefCoord.c
    PrevCoord = NewTarget
    GPCounter = GPCounter + 1
    CNC.RRS.SetNextNamedTarget 0, 0, CartPos, JointPos, Configuration,
0, "GP" & CStr(GPCounter)

        ElseIf ActiveMotionType Like "Circular CW" Or ActiveMotionType
Like "Circular CCW" Then
        DefineMotionType 4
        CircularMotion

End If

End Sub

Public Sub CircularMotion()
Dim ArcRadius As Double
Dim InclineAngle As Double
Dim ArcEndPointDistance As Double
Dim YCoord As Double
Dim XCoord As Double
Dim ArcCenter As RCSCartPos

If Plane = 17 Then
CNC.RRS.SelectMotionType ReturnMotionType()
JointValues DefCoord.x, DefCoord.y, DefCoord.z, DefCoord.b, DefCoord.c
SetFeedrate Feedrate
GPCounter = GPCounter + 1
CNC.RRS.SetNextNamedTarget  0,  0,  CartPos,  JointPos,  Configuration,  0,
"GP" & CStr(GPCounter)

ArcEndPointDistance = Sqr((PrevCoord.x - NewTarget.x) ^ 2 + (PrevCoord.y
- NewTarget.y) ^ 2)
InclineAngle  =  Math.Atn((NewTarget.y  -  PrevCoord.y)  /  (NewTarget.x  -
PrevCoord.x))
XCoord = ArcEndPointDistance / 2

    If CircParam.r <> 0 Then

        If ActiveMotionType = "Circular CW" And CircParam.r > 0 Then
        YCoord = Abs(CircParam.r) - Sqr(CircParam.r ^ 2 - XCoord ^ 2)
        ElseIf ActiveMotionType = "Circular CW" And CircParam.r < 0 Then
        YCoord = Abs(CircParam.r) + Sqr(CircParam.r ^ 2 - XCoord ^ 2)
        ElseIf  ActiveMotionType  =  "Circular  CCW"  And  CircParam.r  >  0
Then
        YCoord = -Abs(CircParam.r) + Sqr(CircParam.r ^ 2 - XCoord ^ 2)
        ElseIf  ActiveMotionType  =  "Circular  CCW"  And  CircParam.r  <  0
Then
        YCoord = -Abs(CircParam.r) - Sqr(CircParam.r ^ 2 - XCoord ^ 2)
        End If

    Else
    Set ArcCenter = GetArcCenter(CircParam.i, CircParam.j, CircParam.k)
    ArcRadius  =  Sqr((PrevCoord.x  -  ArcCenter.px)  ^  2  +  (PrevCoord.y  -
ArcCenter.py) ^ 2)

        If ActiveMotionType = "Circular CCW" Then
        YCoord = -Sqr(ArcRadius ^ 2 - (XCoord - ArcCenter.px) ^ 2)
            ElseIf ActiveMotionType = "Circular CW" Then
            YCoord = Sqr(ArcRadius ^ 2 - (XCoord - ArcCenter.px) ^ 2)
```

```
          End If
      End If

  JointPos.AxesFlags = 0
  CartPos.flag = 3
  CartPos.px = Sqr(PrevCoord.x ^ 2 + PrevCoord.y ^ 2) * Cos(InclineAngle)
  + Sqr(XCoord ^ 2 + YCoord ^ 2) * Cos(Math.Atn(YCoord / XCoord) +
  InclineAngle)
  CartPos.py = Sqr(PrevCoord.x ^ 2 + PrevCoord.y ^ 2) * Sin(InclineAngle)
  + Sqr(XCoord ^ 2 + YCoord ^ 2) * Sin(Math.Atn(YCoord / XCoord) +
  InclineAngle)
  CartPos.pz = PrevCoord.z
  CNC.RRS.SetNextTarget 0, 0, CartPos, JointPos, Configuration, 0

  PrevCoord = NewTarget
  ResetCircParam

  Else
  MsgBox ("Circular motion in other work planes, but XY, is not
  supported")
  End If

  End Sub

  Public Sub ResetCircParam()
  CircParam.r = 0
  CircParam.i = 0
  CircParam.j = 0
  CircParam.k = 0
  End Sub

  Public Sub G27()
  HomeStartSeq
  JointValues DefCoord.x, DefCoord.y, DefCoord.z, DefCoord.b, DefCoord.c
  If (NewTarget.x <> PrevCoord.x Or NewTarget.y <> PrevCoord.y Or
  NewTarget.z <> PrevCoord.z Or NewTarget.b <> PrevCoord.b Or NewTarget.c
  <> PrevCoord.c) Then
  PrevCoord = NewTarget
  CountAndMove
  Else
  GoHome
  End If
  End Sub

  Public Sub G28()
  HomeStartSeq
  JointValues DefCoord.x, DefCoord.y, DefCoord.z, DefCoord.b, DefCoord.c
  If (NewTarget.x <> PrevCoord.x Or NewTarget.y <> PrevCoord.y Or
  NewTarget.z <> PrevCoord.z Or NewTarget.b <> PrevCoord.b Or NewTarget.c
  <> PrevCoord.c) Then
  MemorizeCoordinates NewTarget.x, NewTarget.y, NewTarget.z, NewTarget.b,
  NewTarget.c
  PrevCoord = NewTarget
  CountAndMove
  Else
  GoHome
  End If
  End Sub

  Public Sub G29()
  HomeStartSeq
  JointHomeValues 3
  CountAndMove
  JointValues DefCoord.x, DefCoord.y, DefCoord.z, DefCoord.b, DefCoord.c
```

```
PrevCoord = NewPrimaryHome
CountAndMove
End Sub

Public Sub G30()
HomeStartSeq
JointValues DefCoord.x, DefCoord.y, DefCoord.z, DefCoord.b, DefCoord.c
If  (NewTarget.x  <>  PrevCoord.x  Or  NewTarget.y  <>  PrevCoord.y  Or
NewTarget.z <> PrevCoord.z Or NewTarget.b <> PrevCoord.b Or NewTarget.c
<> PrevCoord.c) Then
PrevCoord = NewPrimaryHome
CountAndMove
End If
JointHomeValues 2
PrevCoord = NewSecondaryHome
CountAndMove
End Sub

Public Sub HomeStartSeq()
CNC.RRS.SelectMotionType 2
DefineMotionType 2
CNC.RRS.SetCartesianPositionSpeed RAPIDFEEDRATE / 60
CNC.RRS.SetCartesianOrientationSpeed 2, MAXROTSPEED / 60
End Sub

Public Sub GoHome()
JointHomeValues 1
PrevCoord = NewPrimaryHome
CountAndMove
End Sub

Public Sub CountAndMove()
GPCounter = GPCounter + 1
CNC.RRS.SetNextNamedTarget  0,  0,  CartPos,  JointPos,  Configuration,  0,
"GP" & CStr(GPCounter)
End Sub

Public Sub ResetPolarAngles()
PolAng17 = 0
PolAng18 = 0
PolAng19 = 0
End Sub

Public Function DegToRad(DegAngle As Double) As Double
DegToRad = (DegAngle / 180) * PI
End Function

Public Sub SetWorkCSParam()

        WorkCS.Props.x = 100
        WorkCS.Props.y = 100
        WorkCS.Props.z = 100
        WorkCS.Props.a = 0
        WorkCS.Props.b = 0
        WorkCS.Props.c = 0
        WorkCS.Gcode = "G54"

End Sub
```

APPENDIX H: DEFAULT MOTION PLANNER SOURCE CODE

```
//DefaultMotion.cpp
#include "stdafx.h"
#include "DefaultMotion.h"
#include "vector3.h"
#include "resource.h"

#include "NumRec.h"
#include "Matrix.h"

// For debugging purposes (sprintf)...
#include <stdio.h>


#define MAX_OBJECT_VELOCITY         1          //Meter's per second
#define MAX_OBJECT_ACCELERATION     100000     //Meter's per second



double sqr( double dValue )
{
      double dRet;

      dRet = dValue * dValue;

      return( dRet );
}


//Helper functions


/*****************************************************************
 * Function:      CalculateAccelerationTime
 *
 * Arguments:     RCS_CAR_POS       vectorOne
 *                      RCS_CAR_POS       vectorTwo
 *
 * Returns:       Distance as a RCS_REAL
 *
 * Created:
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description:
 *
 *
 *****************************************************************/

RCS_REAL    CalculateCartLinearDistance    (RCS_CART_POS    vectorOne,
RCS_CART_POS vectorTwo)
{
      RCS_REAL dDistance;
```

```
        double dX = sqr (vectorTwo.px - vectorOne.px);
        double dY = sqr (vectorTwo.py - vectorOne.py);
        double dZ = sqr (vectorTwo.pz - vectorOne.pz);

        dDistance = sqrt (dX + dY + dZ);

        return (dDistance);
}


/*****************************************************************
 * Function:       CalculateAccelerationTime
 *
 * Arguments:      RCS_REAL    dMaxVelocity
 *                       RCS_REAL    dMaxAcceleration
 *
 * Returns:        Time as a RCS_REAL
 *
 * Created:
 * Author:         Aleksandar Boskovic
 */
/*!
 * Description:
 *
 *
 *****************************************************************/

RCS_REAL CalculateAccelerationTime (RCS_REAL dMaxVelocity,
                                                          RCS_REAL
dMaxAcceleration)
{
        RCS_REAL dTime;

        dTime = dMaxVelocity / dMaxAcceleration;

        return (dTime);
}


/*****************************************************************
 * Function:       CalculateDistanceToMaxVelocity
 *
 * Arguments:      RCS_REAL    dMaxVelocity
 *                       RCS_REAL    dMaxAcceleration
 *
 * Returns:        The distance as a RCS_REAL
 *
 * Created:
 * Author:         Aleksandar Boskovic
 */
/*!
 * Description:   The distance can be found by integrating a velocity-
time
 *                      function.  If we make quite a few assumptions
such as constant
 *                      acceleration and an initial velocity of zero,
then by integrate
 *                      v(t) = [a*v] dt, from 0 to t and then substitute
t = v/a, we
 *                      obtain the equation as listed below.
 *
 *****************************************************************/
```

```
RCS_REAL CalculateDistanceToMaxVelocity (RCS_REAL dMaxVelocity, RCS_REAL
dMaxAcceleration)
{
      RCS_REAL dDistance;

      dDistance = sqr (dMaxVelocity) / 2 / dMaxAcceleration;

      return (dDistance);
}


/*****************************************************************
 * Function:      CalculateTimeAtMaximumVelocity
 *
 * Arguments:     RCS_REAL    dTotalDistance
 *                      RCS_REAL    dMaxVelocity
 *                      RCS_REAL    dMaxAcceleration
 *
 * Returns:       Time as a RCS_REAL
 *
 * Created:
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description: Here we simply subtract the acceleration and
 *                      deceleration distance from total distance.  Then
 *                      the remaining distance is the portion at the max
 *                      velocity.
 *
 *
 *****************************************************************/

RCS_REAL CalculateTimeAtMaximumVelocity (RCS_REAL dTotalDistance,
                                                              RCS_REAL
dMaxVelocity,
                                                              RCS_REAL
dMaxAcceleration)
{
      RCS_REAL    dDistToMaxVel,
                        dTimeAtMaxVel;

      dDistToMaxVel   =   CalculateDistanceToMaxVelocity   (dMaxVelocity,
dMaxAcceleration);

      dTimeAtMaxVel   =   (dTotalDistance   -   2   *   dDistToMaxVel)   /
dMaxVelocity;

      return (dTimeAtMaxVel);
}


/*****************************************************************
 * Function:      CalculateChangeInPositions
 *
 * Arguments:     RCS_CART_POS      P1
 *                      RCS_CART_POS      P2
 *                      RCS_CART_POS      &Change
 *
 * Returns:       bool (at this point its always true)
 *
 * Created:
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description:
 *****************************************************************/

bool CalculateChangeInPositions (RCS_CART_POS P1,
                                                   RCS_CART_POS P2,
                                                   RCS_CART_POS &Change)
{
      Change.CartPosFlag = 0;

      Change.ax = P1.ax - P2.ax;
      Change.ay = P1.ay - P2.ay;
      Change.az = P1.az - P2.az;

      Change.nx = P1.nx - P2.nx;
      Change.ny = P1.ny - P2.ny;
      Change.nz = P1.nz - P2.nz;

      Change.ox = P1.ox - P2.ox;
      Change.oy = P1.oy - P2.oy;
      Change.oz = P1.oz - P2.oz;

      Change.px = P1.px - P2.px;
      Change.py = P1.py - P2.py;
      Change.pz = P1.pz - P2.pz;

      return true;
}


/*****************************************************************
 * Function:      FindBiggestChange
 *
 * Arguments:
 *                      double                    x
 *                      double                    y
 *                      double                    z
 *
 * Returns:       double
 *
 * Created:       21-Nov-99
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description: First we need to explain why we choose such arguments.
 *                      We pass in a vector by an array of three
doubles.  We never
 *                      choose a CVector3 (our code) or whatever in acis
since we are
 *                      only interested in the positions, and thats it.
No need to
 *                      for workspace to create and destory some objects
when it isn't
 *                      really all to necessary.  True it doesn't add to
much overhead,
 *                      but we are not going to worry to much about that
right now.
 *
 *                      First we need to find the amount of time needed
for the longest
 *                      possible move.  Then we find the amount of time
according to the
 *                      longest time.
 *
 *****************************************************************/

double FindBiggestChange (double x, double y, double z)
{
      double largest_distance;

      x > y
            ? (z > x ? largest_distance = z : largest_distance = x)
            : (z > y ? largest_distance = z : largest_distance = y);

      return largest_distance;
}



//////////////////////////////////////////////////////////////////////
/////////////
//////////////////////////////////////////////////////////////////////
/////////////
//    MOTION PLANNER CLASS
//////////////////////////////////////////////////////////////////////
/////////////

//Class member functions

/*****************************************************************
 * Function:      CDefaultMotionPlanner::CDefaultMotionPlanner
 *
 * Arguments:
 *    void
 *
 * Created:       990806
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description:   Standard constructor
 *
 *****************************************************************/
CDefaultMotionPlanner::CDefaultMotionPlanner(int planner_type )
{
      m_dArcStartAngle = 0.0;

      // Assume motion planner is attached to an object
      // Kinematics pointer will be intialised later
      m_pKinematics = NULL;
      m_pfnGetInverseKinematic = NULL;
      m_pDefaultShell = NULL;

      m_pScaledJointMotion = NULL;

      //Set   up   a   linked   list   to   store   targets   specifed   by
Set_Next_Target calls
      m_TargetsList.InitializeList();

      //Set default values
      m_nMotionType = MOTION_LINEAR;

      m_nInterpolationTime = DEFAULT_INTERPOLATION_TIME;

      m_nTrajectoryMode = 1;  //targets belong to same trajectory (not
supported)

      m_nFlyByMode = 0;       //off
      m_nCurrentInterval = 0; //internal    marker    for    calls    to
'Get_Next_Step'

      m_nCurrentTarget = 0;   //Index into m_TargetsList, showing which
target the
                                          //robot is heading towards

      m_nRobotStatus = 0;          //Robot is stationary and awaiting
target

      // set up for defaults for orientation interpolation
      m_dMaxOrientationAngleVel = 4.0 ;
      m_dMaxOrientationAngleAcc = 8.0 ;
      m_nInterpolationMode = 3 ;
      m_Txform.SetOrientationType( orientWPR ) ;
      m_DoOrientation = TRUE ;

      m_cartMotionStart_Position.CartPosFlag = 0;
      m_jointMotionStart_Position.AxesFlags  = 0;
      m_jointLast_Position.AxesFlags = 0;

      m_pTempTarget = NULL;

      if (planner_type == DMP_OBJECT)
            m_bObject = true;
      else
            m_bObject = false;

      // The default velocity of for Object Motion is 1000 mm/sec
      m_dMaxObjVel = 100;

      ZeroMemory( m_JointMotion, sizeof(m_JointMotion) );
      ZeroMemory( m_JointMotionCurrent, sizeof(m_JointMotionCurrent) );

      m_dMaxLinearVelLimit = 0.0;
      m_dMaxLinearAccLimit = 0.0;
      m_dMaxLinearVelCurrent = 0.0;
      m_dMaxLinearAccCurrent = 0.0;
      m_dMaxLinearVel = 0.0;
      m_dMaxLinearAcc = 0.0;
}


/*******************************************************************
 *
 * Function:      CDefaultMotionPlanner::~CDefaultMotionPlanner
 *
 * Arguments:
 *
 *
 * Returns:
 *
 *
 * Created:       990713
 *
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description:   standard destructor
 *
 *
 * Comments:
```

```
 *
 ***************************************************************/
CDefaultMotionPlanner::~CDefaultMotionPlanner()
{
      //Remove any profiles for joint co-ordinated movement
      if( NULL != m_pScaledJointMotion )
      {
            delete [] m_pScaledJointMotion;
            m_pScaledJointMotion = NULL;
      }

      //Remove any targets from the list
      m_TargetsList.DeInitializeList();

      // Remove any messages
      ResetMessages( );

}


RCS_INT     CDefaultMotionPlanner::Select_FlyBy_Mode( RCS_INT FlyByOn )
{
      m_nFlyByMode = FlyByOn;

      return( 0 ); //Success
}



RCS_INT     CDefaultMotionPlanner::Set_Interpolation_Time(      RCS_REAL
InterpolationTime )
{
      m_nInterpolationTime = InterpolationTime;

      return( 0 ); //Success
}




RCS_INT     CDefaultMotionPlanner::Select_Trajectory_Mode(       RCS_INT
TrajectoryOn )
{
      m_nTrajectoryMode = TrajectoryOn;

      return( 0 ); //Success
}




RCS_INT     CDefaultMotionPlanner::Select_Dominant_Interpolation
(RCS_INT DominantIntType,

                        RCS_INT DominantIntParam)
{
      // Right now we are only going to allow for the position to be
master (1), or the
      // orientation to be master (2), or automatic (4).  We are by
default using
      // automatic mode which tries the position first then the
orientation.

      switch (DominantIntType)
```

```
        {
              case 1:
              case 2:
              case 4: m_nDominantInterpolationType = DominantIntType;
                           return RRS_OK;
        }

        return RRS_INTERPOL_SPACE_NOT_SUPPORTED;
}



/*******************************************************************
 *
 * Function:       CDefaultMotionPlanner::Select_Motion_Type
 *
 * Arguments:
 *    RCS_INT MotionType
 *
 *
 * Returns:        RCS_INT
 *
 *
 * Created:        990624
 *
 * Author:         Aleksandar Boskovic
 */
/*!
 * Description:   Sets the motion type for the default motion planner
 *
 *
 * Comments:      Added extra types to support G-Code circular motion
 *
 ******************************************************************/
RCS_INT CDefaultMotionPlanner::Select_Motion_Type( RCS_INT MotionType )
{
        RCS_INT            nRet = 0;

        switch( MotionType )
        {
              case MOTION_JOINT:
              case MOTION_LINEAR:
              case MOTION_CIRCULAR:
              case MOTION_CIRCULAR_CLOCKWISE:
              case MOTION_CIRCULAR_ANTICLOCKWISE:
              {
                    m_nMotionType = MotionType;
              }
              break;

              default:
              {
                    nRet = RRS_MOTION_TYPE_NOT_SUPPORTED;
              }
              break;
        }

        return nRet;
}
```

APPENDIX I: G-CODE OFF-LINE PROGRAMMING SOURCE CODE

```
// PathToGCode.cpp: implementation of the CPathToGCode class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "workspace.h"
#include "PathToGCode.h"
#include "Path.h"
#include "GP.h"
#include "GPAction.h"
#include "WorkspaceDoc.h"
#include "Find.h"
#include "CADUtils.h"
#include "MathUtil.h"
#include "Vector3.h"
#include "constrct/kernapi/api/cstrapi.hxx"
#include "kernel/geomhusk/geom_utl.hxx"
#include "kernel\geomhusk\curveq.hxx"
#include "GCodePointAccuracy.h"
#include "AcisMFC\acisdoc.hxx"
#include "acismfc\amfc_utl.hxx"
#include "AcisMFC\Tools\tools.hxx"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////


/*****************************************************************
 *
 * Function:      CPathToGCode::CPathToGCode
 *
 * Arguments:
 *    LPCTSTR     lpszProgramFile,
 *  CRobot  *pRobot,
 *  CPath   *pPath
 *
 *
 *
 * Returns:
 *
 *
 * Created:       990718
 *
 * Author:        Aleksandar Boskovic
 */
/*!
```

```
 * Description:
 *
 *
 * Comments:      Constructor for class which converts a path into
 *                      the GCode language.
 *
 *****************************************************************/
CPathToGCode::CPathToGCode(LPCTSTR  lpszProgramFile,
                                       CRobot   *pRobot,
                                       CPath    *pPath)
:CPathToLanguage( lpszProgramFile, pRobot, pPath )
{
      m_strLanguage = "GCODE";
      m_pHeadPath = pPath;
      m_bVerbose=false;
      m_bFeedEmitted=false;
      m_bXEmitted=false;
      m_bYEmitted=false;
      m_bZEmitted=false;
      m_bBEmitted=false;
      m_bCEmitted=false;
      m_bOpCodeEmitted=false;
      m_bPlaneEmitted=false;


}


/*****************************************************************
 *
 * Function:      CPathToGCode::WritePath
 *
 * Arguments:
 *
 *
 * Returns:       CString
 *
 *
 * Created:       990718
 *
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description:
 *
 *
 * Comments:      Writes the path
 *
 *****************************************************************/
CString CPathToGCode::WritePath()
{
      CStdioFile  ProgramFile(
                              m_strFileName,
                              CFile::modeCreate | CFile::modeWrite );
      // Write sub-paths
      WriteFunction( m_pPath, ProgramFile );

      return CPathToLanguage::WritePath();
}

void CPathToGCode::StartFunction(CPath *pPath, CStdioFile &rFile)
{
      m_bFeedEmitted=false;
      m_bXEmitted=false;
```

```
       m_bYEmitted=false;
       m_bZEmitted=false;
       m_bBEmitted=false;
       m_bCEmitted=false;
       m_bOpCodeEmitted=false;
       m_bPlaneEmitted=false;
}

/*****************************************************************
 *
 * Function:      CPathToGCode::WriteFunction
 *
 * Arguments:
 *     CPath *pPath,
 *    CStdioFile &rFile
 *
 *
 * Returns:       void
 *
 *
 * Created:       991007
 *
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description:
 *
 *
 * Comments:
 *
 *****************************************************************/
void CPathToGCode::WriteFunction( CPath *pPath, CStdioFile &rFile )
{
      CWorkspaceDoc     *pDoc = GetActiveDoc();
      CString strLine;

      StartFunction(pPath, rFile);
      if (!pPath->GetPathParent())
      {
            // Write procedure to call sub-paths
            strLine.Format( "\nO%s",FindGCodeName(m_pPath->GetName()));

            rFile.WriteString( strLine );

            // Write Header information

            strLine.Format("\n%s D1 H1", m_LineNo.Next());
            rFile.WriteString( strLine);

            strLine.Format("\n%s G92.1", m_LineNo.Next() );
            rFile.WriteString( strLine);

            strLine.Format("\n%s   G00   G17   G40   G80   G90   G54   G94",
m_LineNo.Next() );
            rFile.WriteString( strLine);

            strLine.Format("\n%s G92 X0.000 Y0.000 Z0.000 B0.000 C0.000
F250.000", m_LineNo.Next() );
            rFile.WriteString( strLine);
      }
      else
      {
            strLine.Format( "\nO%s ",FindGCodeName(pPath->GetName()));
            rFile.WriteString( strLine );

            strLine.Format("\n%s F250", m_LineNo.Next());
      }


            // Loop through target list
      CPtrList    *pTargetList = pPath->GetTargetList();
      POSITION posList = pTargetList->GetHeadPosition();

      while( posList )
      {
            // Get the name of the next target
            CPathTarget   *pPT   =   (CPathTarget*)pTargetList->GetNext(
posList );

            if (pPT->isPath())
            {
                  // This target is a path

                  // Should already have written the procedure for
                  // this sub-path, so just make a call to it
                  strLine.Format(                                 "\n%s
M98P%s",m_LineNo.Next(),FindGCodeName(pPT->GetName()));
                  rFile.WriteString( strLine );
            }
            else
            {
                  if (pPT->isGPMove())
                  // This target is a GPMove
                        WriteMove( (CGPMove*)pPT, rFile );
            }

      }

      if(!pPath->GetPathParent())
            strLine.Format(  "\n%s  G92.1\n%s  M02"  ,  m_LineNo.Next(),
m_LineNo.Next() );
      else
            strLine.Format( "\n%s M99", m_LineNo.Next()  );
      rFile.WriteString( strLine );

      // Loop through sub-paths
      CPtrList    *pPathList = pPath->GetTargetList();
      posList = pPathList->GetHeadPosition();

      while( posList )
      {
            CPathTarget *pPT;

            pPT =  (CPathTarget*)pPathList->GetNext( posList );
            if (pPT->isPath())
            {
                  CString strSubFuncPath = rFile.GetFilePath();
                  CString strMasterFuncName = rFile.GetFileName();
                  strSubFuncPath.Replace(strMasterFuncName,m_pHeadPath-
>GetName()+"_O"+FindGCodeName(((CPath*)pPT)->GetName())+".ppg");
                  CStdioFile  subFile(
                              strSubFuncPath,
                              CFile::modeCreate | CFile::modeWrite );
                  StartFunction(pPath, subFile);

                  WriteFunction( ((CPath*)pPT), subFile );
            }
      }
}

double PosZero(double dVal)
{
      if (AlmostZero(dVal))
            return 0.0;
      else
            return dVal;
}

/*****************************************************************
 *
 * Function:      CPathToGCode::WriteMove
 *
 * Arguments:
 *     CGPMove* pGPMove,
 *    CStdioFile &rFile
 *
 *
 * Returns:       void
 *
 *
 * Created:       991007
 *
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description:
 *
 *
 * Comments:
 *
 *****************************************************************/
void CPathToGCode::WriteMove( CGPMove* pGPMove, CStdioFile &rFile )
{
      static bool bConfJ = true;
      static double dPrevPos[3]={0.0,0.0,0.0};  // world co-ords set up
as start pos
      static double dPrevB=0.0;
      static double dPrevC=0.0;
      static double dPrevFeed=250.0;
      static double dI=0.0, dJ=0.0, dK=0.0;
      int nActions;

      CString strLocPic="%.3f";
      static CCutDownGP *pPrevMiniGP=NULL;
      static CString strPrevOpCode="";
      static CString strPrevPlane="";
      static CString strPrevCircDir="";
      static CString strPrevX="", strPrevY="", strPrevZ="", strPrevB="",
strPrevC="";
      static CString strPrevFeed;
      static CString strPrevI="",strPrevJ="",strPrevK="";
      bool  bOpCodeReqd=true,
                  bPlaneReqd=true,
                  bXReqd=true,
                  bYReqd=true,
                  bZReqd=true,
                  bBReqd=true,
                  bCReqd=true,
                  bIReqd=true,
                  bJReqd=true,
                  bKReqd=true,
                        bFeedReqd=true,
                        bNonOrtho=false;

        CString           strLine;
        //transf          transformTP;

        //transf  pGPPos = pGPMove->GetGP()->GetTransf();
        RCS_JOINT_POS JointTemp;

        CCutDownGP   *   pMiniGP   =   GetMiniGPFromMap(pGPMove->GetGP()-
>GetName());
                if(! pMiniGP )
                {
                        AfxMessageBox("MiniGP not found");
                }

        JointTemp = pMiniGP->GetJointPos();
        //transformTP   =   m_pRobot->ConvertGPtoTP(   pGPMove->GetWCS(   )-
>to_model( ) );

        //RCS_FRAME frRCS;

        //ConvertTransfToRCSFrame( frRCS, transformTP );

        if     (m_bXEmitted     &&     AlmostZero(JointTemp.AxesValues[0]-
dPrevPos[0]))
                bXReqd = m_bVerbose;
        else
        {
                strPrevX.Format("                                X"+strLocPic,
PosZero(JointTemp.AxesValues[0]));
        }

        if     (m_bYEmitted     &&     AlmostZero(JointTemp.AxesValues[1]-
dPrevPos[1]))
                bYReqd = m_bVerbose;
        else
        {
                strPrevY.Format("                                Y"+strLocPic,
PosZero(JointTemp.AxesValues[1]));
        }

        if     (m_bZEmitted     &&     AlmostZero(JointTemp.AxesValues[2]-
dPrevPos[2]))
                bZReqd = m_bVerbose;
        else
        {
                strPrevZ.Format("                                Z"+strLocPic,
PosZero(JointTemp.AxesValues[2]));
        }

        if (m_bBEmitted && AlmostZero(JointTemp.AxesValues[3]-dPrevB))
                bBReqd = m_bVerbose;
        else
        {
                strPrevB.Format("                                B"+strLocPic,
PosZero(RadToDeg(JointTemp.AxesValues[3])));
        }

        if (m_bCEmitted && AlmostZero(JointTemp.AxesValues[4]-dPrevC))
                bCReqd = m_bVerbose;
        else
        {
                strPrevC.Format("                    C"+strLocPic,
PosZero(RadToDeg(JointTemp.AxesValues[4])));
        }

        if (pGPMove->HasPosSpeed())
        {
                if  (m_bFeedEmitted  &&  AlmostZero(pGPMove->GetPosSpeed()-
dPrevFeed))
                        bFeedReqd = m_bVerbose;
                else
                {
                        strPrevFeed.Format(   "   F"+strLocPic,   pGPMove-
>GetPosSpeed());
                }
        }
        else
        {
                strPrevFeed.Format( " F"+strLocPic, dPrevFeed);
                bFeedReqd  =  m_bVerbose  ||  !m_bFeedEmitted;  //  ???  repeat
last feed rate if verbose?
        }

        if( pGPMove->GetMotionType() == "Circular" )
        {
                double dRadius;
                position posCenter;
                unit_vector vNormal;
                bool bDir;

                ASSERT(pPrevMiniGP);
                GetCircleParams(pPrevMiniGP,   pGPMove,   bDir,   posCenter,
vNormal, dRadius);

                if (bDir)
                {
                        if (strPrevOpCode == " G02")
                                bOpCodeReqd = m_bVerbose || !m_bOpCodeEmitted;
                        else
                                strPrevOpCode=" G02";
                }
                else
                {
                        if (strPrevOpCode == " G03")
                                bOpCodeReqd = m_bVerbose || !m_bOpCodeEmitted;
                        else
                        {
                                strPrevOpCode=" G03";
                        }
                }
                if (AlmostZero(JointTemp.AxesValues[2]-dPrevPos[2]))
                {
                        if (strPrevPlane == " G17")
                                bPlaneReqd = m_bVerbose || m_bPlaneEmitted;
                        else
                                strPrevPlane = " G17";
                        dI = posCenter.x() - dPrevPos[0];
                        dJ = posCenter.y() - dPrevPos[1];
                        strPrevI.Format(" I"+strLocPic, dI);
                        strPrevJ.Format(" J"+strLocPic, dJ);
                        bZReqd = false;
                        bKReqd = false;
                }
                else
                if (AlmostZero(JointTemp.AxesValues[1]- dPrevPos[1]))
```

```
            {
                  if (strPrevPlane == " G18")
                        bPlaneReqd = m_bVerbose || m_bPlaneEmitted;
                  else
                        strPrevPlane = " G18";
                  bYReqd = false;
                  dI = posCenter.x() - dPrevPos[0];
                  dK = posCenter.z() - dPrevPos[2];
                  strPrevI.Format(" I"+strLocPic, dI);
                  strPrevK.Format(" K"+strLocPic, dK);
                  bJReqd = false;
            }
            else
            if (AlmostZero(JointTemp.AxesValues[0]-dPrevPos[0]))
            {
                  if (strPrevPlane == " G19")
                        bPlaneReqd = m_bVerbose || m_bPlaneEmitted;
                  else
                        strPrevPlane = " G19";
                  bXReqd = false;
                  dJ = posCenter.y() - dPrevPos[1];
                  dK = posCenter.z() - dPrevPos[2];
                  strPrevJ.Format(" J"+strLocPic, dJ);
                  strPrevK.Format(" K"+strLocPic, dK);
                  bIReqd = false;
            }
            else
            {
                  bNonOrtho = true;
            }

            if (bNonOrtho)
                  strLine.Format("\n(non-ortho   circular   moves   not
supported yet)");
            else
            {
                  strLine.Format( "\n%s%s%s%s%s%s%s%s%s%s%s",
                  m_LineNo.Next(),
                  bPlaneReqd  ?     strPrevPlane      :"",
                  bOpCodeReqd ?     strPrevOpCode     :"",
                  bXReqd            ?     strPrevX          :"",
                  bYReqd            ?     strPrevY          :"",
                  bBReqd            ?     strPrevB          :"",
                  bCReqd            ?     strPrevC          :"",
                  bIReqd            ?     strPrevI          :"",
                  bJReqd            ?     strPrevJ          :"",
                  bKReqd            ?     strPrevK          :"",
                  bFeedReqd   ?     strPrevFeed       :""
                  );
                  m_bXEmitted = m_bXEmitted || bXReqd;
                  m_bYEmitted = m_bYEmitted || bYReqd;
                  m_bZEmitted = m_bZEmitted || bZReqd;
                  m_bBEmitted = m_bBEmitted || bBReqd;
                  m_bCEmitted = m_bCEmitted || bCReqd;
                  m_bPlaneEmitted = m_bPlaneEmitted || bPlaneReqd;
                  m_bOpCodeEmitted = m_bOpCodeEmitted || bOpCodeReqd;
                  m_bFeedEmitted = m_bFeedEmitted || bFeedReqd;
            }


      }
      else if( pGPMove->GetMotionType() == "Joint" )
      {
            if (strPrevOpCode == " G00")
```

```
                bOpCodeReqd = m_bVerbose || !m_bOpCodeEmitted;
            else
                strPrevOpCode = " G00";
            strLine.Format( "\n%s%s%s%s%s%s%s",
                m_LineNo.Next(),
                bOpCodeReqd ?      strPrevOpCode      :"",
                bXReqd             ?     strPrevX           :"",
                bYReqd             ?     strPrevY           :"",
                bZReqd             ?     strPrevZ           :"",
                bBReqd             ?     strPrevB           :"",
                bCReqd             ?     strPrevC           :""
                );
            m_bOpCodeEmitted = m_bOpCodeEmitted || bOpCodeReqd;
            m_bXEmitted = m_bXEmitted || bXReqd;
            m_bYEmitted = m_bYEmitted || bYReqd;
            m_bZEmitted = m_bZEmitted || bZReqd;
            m_bBEmitted = m_bBEmitted || bBReqd;
            m_bCEmitted = m_bCEmitted || bCReqd;
        }
        else if( pGPMove->GetMotionType() == "Linear" )
        {
            if (strPrevOpCode == " G01")
                bOpCodeReqd = m_bVerbose || !m_bOpCodeEmitted;
            else
                strPrevOpCode = " G01";
            strLine.Format( "\n%s%s%s%s%s%s%s%s",
                m_LineNo.Next(),
                bOpCodeReqd ?      strPrevOpCode      :"",
                bXReqd             ?     strPrevX           :"",
                bYReqd             ?     strPrevY           :"",
                bZReqd             ?     strPrevZ           :"",
                bBReqd             ?     strPrevB           :"",
                bCReqd             ?     strPrevC           :"",
                bFeedReqd   ?      strPrevFeed        :""
                );
            m_bOpCodeEmitted = m_bOpCodeEmitted || bOpCodeReqd;
            m_bXEmitted = m_bXEmitted || bXReqd;
            m_bYEmitted = m_bYEmitted || bYReqd;
            m_bZEmitted = m_bZEmitted || bZReqd;
            m_bBEmitted = m_bBEmitted || bBReqd;
            m_bCEmitted = m_bCEmitted || bCReqd;
            m_bFeedEmitted = m_bFeedEmitted || bFeedReqd;
        }

        rFile.WriteString( strLine );


        // Write tool action as GCODE comment
        if((nActions =  pGPMove->GetNumGPActions()))
        {
            for (int i=0;i<nActions;i++)
            {
                CString    strAction    =    pGPMove->GetGPAction(i)-
>GetDescription();
                if (strAction == "WaterJetOn")
                    strLine.Format("\n%s M08", m_LineNo.Next());
                else if (strAction == "WaterJetOff")
                    strLine.Format("\n%s M09", m_LineNo.Next());
                else if (strAction == "AbrasiveOn")
                    strLine.Format("\n%s M10", m_LineNo.Next());
                else if (strAction == "AbrasiveOff")
                    strLine.Format("\n%s M11", m_LineNo.Next());
                else
                    strLine.Format( "\n(- %s);", strAction );
```

```
                  rFile.WriteString( strLine );
            }
      }
      pPrevMiniGP = pMiniGP;
      dPrevPos[0] = JointTemp.AxesValues[0];
      dPrevPos[1] = JointTemp.AxesValues[1];
      dPrevPos[2] = JointTemp.AxesValues[2];
      dPrevB = JointTemp.AxesValues[3];
      dPrevC = JointTemp.AxesValues[4];

      if (pGPMove->HasPosSpeed())
            dPrevFeed = pGPMove->GetPosSpeed();
}


/*****************************************************************
 *
 * Function:      CPathToGCode::~CPathToGCode
 *
 * Arguments:
 *
 *
 * Returns:
 *
 *
 * Created:       991007
 *
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description:
 *
 *
 * Comments:
 *
 *****************************************************************/
CPathToGCode::~CPathToGCode()
{

}


/*****************************************************************
 *
 * Function:      CPathToGCode::WriteGCODEFrames
 *
 * Arguments:
 *    CStdioFile &rFile,
 *    CRobot *pRobot
 *
 *
 * Returns:       void
 *
 *
 * Created:       991007
 *
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description:
 *
 *
 * Comments:
 *
 ***************************************************************/
void CPathToGCode::WriteGCODEFrames(CStdioFile &rFile, CRobot *pRobot)
{

}


/****************************************************************
 *
 * Function:      CPathToGCode::FindGCodeName
 *
 * Arguments:
 *    CString sName
 *
 *
 * Returns:       CString
 *
 *
 * Created:       991007
 *
 * Author:        Aleksandar Boskovic
 */
/*!
 * Description:
 *
 *
 * Comments:
 *
 ***************************************************************/
CString CPathToGCode::FindGCodeName(CString sPathName)
{
      static CString rVal;

      if (!m_PathNameMap.Lookup(sPathName, rVal))
      {
            rVal.Format("%04d", m_PathNameMap.GetCount()+1);
            m_PathNameMap.SetAt(sPathName, rVal);
      }

      return rVal;
}

void CPathToGCode::SetStateCircDir(bool bCircDir)
{
      m_bCircDir = bCircDir;
}

bool CPathToGCode::GetStateCircDir(void)
{
      return m_bCircDir;
}

void CPathToGCode::SetStatePos(double aPos[])
{
      m_aPos[0] = aPos[0];
      m_aPos[1] = aPos[1];
      m_aPos[2] = aPos[2];
}

void CPathToGCode::GetStatePos(double aPos[])
{
      aPos[0] = m_aPos[0];
      aPos[1] = m_aPos[1];
      aPos[2] = m_aPos[2];
```

```
}

void CPathToGCode::SetStateOrient(double aOrient[])
{
      m_aOrient[0] = aOrient[0];
      m_aOrient[1] = aOrient[1];
}

void CPathToGCode::GetStateOrient(double aOrient[])
{
      aOrient[0] = m_aOrient[0];
      aOrient[1] = m_aOrient[1];
}

void CalcMidPoint(position pStartWCS,position pEndWCS,position pMidpt);

bool  CPathToGCode::GetCircleParams(CCutDownGP  *pMiniStartGP,  CGPMove*
pEndGP, bool &bDir, position &posCenter, unit_vector &vNormal, double
&dRadius)
{
      CCutDownGP   *   pMiniEndGP   =   GetMiniGPFromMap(pEndGP->GetGP()-
>GetName());
            if(! pMiniEndGP )
            {
                  AfxMessageBox("MiniEndGP not found");
            }


      CCutDownGP   *   pMiniViaGP   =   GetMiniGPFromMap(pEndGP->GetViaGP()-
>GetName());
            if(! pMiniViaGP )
            {
                  AfxMessageBox("MiniViaGP not found");
            }

      //WCS *pWCSStart, *pWCSEnd, *pWCSMid;
      EDGE *pEdge;
      outcome result;
      double dDist, dDist2;
      CVector3 vecEndPoint, m_ArcStartPoint, posVia;
      double    m_dAngleAboutX,    m_dAngleAboutY,    m_dAngleAboutZ;//
m_dArcTotalAngle, theta;
      //double DisFromAxis;
      position m_ArcCentre;
      bool bNegXAxis;
      RCS_JOINT_POS JointEnd=pMiniEndGP->GetJointPos();
      RCS_JOINT_POS JointStart=pMiniStartGP->GetJointPos();
      RCS_JOINT_POS JointMid=pMiniViaGP->GetJointPos();
      position posEnd(JointEnd.AxesValues);
      position posStart(JointStart.AxesValues);
      position posMid(JointMid.AxesValues);

      if (pEndGP->GetMotionType() != "Circular")
            return false;

      //pWCSStart = pStartGP->GetWCS();
      //pWCSMid = pEndGP->GetViaGP()->GetWCS();
      //pWCSEnd = pEndGP->GetWCS();
      dDist = distance_to_point(posStart,posEnd);
      dDist2 = distance_to_point(posStart,posMid);
      dDist2 = distance_to_point(posEnd,posMid);
            if (AlmostZero(dDist))
            {
                  position pMidpt;
                  CalcMidPoint(posStart,posMid,pMidpt);

                  result = api_curve_arc_center_edge (
                        pMidpt,
                        posStart,
                        posMid,
                        NULL, // &pWCSStart->z_axis(),
                        pEdge);
            }
            else
                  result = api_curve_arc_3pt(
                        posStart,
                        posMid,
                        posEnd,
                        FALSE,
                        pEdge );
                  CheckOutcome(result);

      if (result.ok())
      {
            get_curve_radius( pEdge, dRadius );
            get_curve_center( pEdge, posCenter );
            get_curve_normal( pEdge, vNormal );
            pEdge->lose();
      }
      else
            return false;
      m_dAngleAboutX = m_dAngleAboutY = m_dAngleAboutZ = 0.0;

      vecEndPoint.SetX(posEnd.x());
      vecEndPoint.SetY(posEnd.y());
      vecEndPoint.SetZ(posEnd.z());

      m_ArcStartPoint.SetX(posStart.x());
      m_ArcStartPoint.SetY(posStart.y());
      m_ArcStartPoint.SetZ(posStart.z());
      posVia.SetX(posMid.x());
      posVia.SetY(posMid.y());
      posVia.SetZ(posMid.z());

      vecEndPoint = vecEndPoint - m_ArcStartPoint;
      m_ArcStartPoint = m_ArcStartPoint - vecEndPoint;

      if (vecEndPoint.GetZ ())
      {
            // Rotate the END point about y axis to bring point onto xy
plane
            m_dAngleAboutY     =     atan2     (-vecEndPoint.GetZ     (),
vecEndPoint.GetX ());
            vecEndPoint.RotateAboutY (-m_dAngleAboutY);
            posVia.RotateAboutY(-m_dAngleAboutY);
      }

      if (vecEndPoint.GetX() < 0)
            bNegXAxis = true;
      else
            bNegXAxis = false;
      if (vecEndPoint.GetY ())
      {
            // Rotate about z axis to bring the END point onto x axis
            m_dAngleAboutZ     =     atan2     (vecEndPoint.GetY     (),
vecEndPoint.GetX ());
            vecEndPoint.RotateAboutZ (-m_dAngleAboutZ);
```

```
              posVia.RotateAboutY(-m_dAngleAboutZ);
        }

        if (posVia.GetY() > 0)
              bDir = !bNegXAxis;
        else
              bDir = bNegXAxis;

        return true;


  }
```

APPENDIX J: THE TEST G-CODE PART PROGRAM GENERATED OFF-LINE

```
%
O0001
N00005 D1 H1
N00010 G92.1
N00015 G00 G17 G40 G80 G90 G54 G94
N00020 G92 X0.000 Y0.000 Z0.000 B0.000 C0.000 F250.000
N00025 M98P0002
N00030 M98P0003
N00035 M98P0004
N00045 G92.1
N00040 M02
%


%
O0002
N00055 G01 X500.000 Y284.100 Z52.000 B-180.000 C0.000 F250.000
N00060 M08
N00065 G17 G02 Y715.900 I0.000 J215.900
N00070 G17 X493.218 Y284.207 B178.200 I0.000 J-215.900
N00075 G01 X500.000 Y284.100 B-180.000
N00080 M09
N00085 M99
%


%
O0003
N00095 G01 X519.050 Y583.564 Z52.000 B134.149 C67.764 F250.000
N00100 M08
N00105 Y636.859 B139.081 C67.511
N00110 G02 X540.365 Y646.957 B74.641 C0.000 I14.807 J-3.710
N00115 X647.067 Y539.962 B15.202 I-40.365 J-146.957
N00120 G03 X637.156 Y519.050 B32.378 C70.966 I-12.226 J-7.010
N00125 G01 X584.365 B-320.167 C68.559
N00130 G03 X569.355 Y531.565 B114.472 C0.000 I2.938 J18.783
N00135 X531.714 Y569.287 B155.405 I-69.355 J-31.565
N00140 X519.050 Y583.564 B134.149 C67.764 I6.343 J18.382
N00145 M09
N00150 G01 X416.436 Y519.050 B-130.116 C67.542
N00155 M08
N00160 X363.141 B-134.182 C67.595
N00165 G03 X353.043 Y540.365 B164.641 C0.000 I3.710 J14.807
N00170 G02 X460.038 Y647.067 B105.202 I146.957 J-40.365
N00175 X480.950 Y636.600 B41.727 C-67.500 I5.945 J-14.247
N00180 G01 Y584.365
N00185 G03 X468.435 Y569.355 B-155.528 C0.000 I-19.023 J3.138
N00190 G02 X430.713 Y531.714 B-114.595 I31.565 J-69.355
N00195 G03 X416.436 Y519.050 B-130.116 C67.542 I-16.305 J4.002
N00200 M09
N00205 G01 X480.950 Y412.172 B221.727 C-67.500
N00210 M08
```

```
N00215 Y363.141 B-138.273
N00220 G03 X459.635 Y353.043 B-105.359 C0.000 I-14.807 J3.710
N00225 X352.933 Y460.038 B-164.798 I40.365 J146.957
N00230 G02 X363.400 Y480.950 B131.727 C-67.500 I14.068 J6.035
N00235 G01 X415.635
N00240 G02 X430.645 Y468.435 B-65.528 C0.000 I-3.138 J-19.023
N00245 X468.286 Y430.713 B-24.595 I69.355 J31.565
N00250 X480.950 Y412.172 B221.727 C-67.500 I-5.466 J-17.329
N00255 M09
N00260 G01 X583.564 Y480.950 B48.273 C67.500
N00265 M08
N00270 X636.859
N00275 G02 X646.957 Y459.635 B-15.359 C0.000 I-3.710 J-14.807
N00280 X539.962 Y352.933 B-74.798 I-146.957 J40.365
N00285 G03 X519.050 Y362.844 B318.273 C67.500 I-5.416 J15.591
N00290 G01 Y415.635 B-41.727
N00295 G02 X531.565 Y430.645 B24.472 C0.000 I20.363 J-4.255
N00300 X569.287 Y468.286 B65.405 I-31.565 J69.355
N00305 X583.564 Y480.950 B48.273 C67.500 I18.382 J-6.343
N00310 M09
N00315 M99
%


%
O0004
N00325 G01 X500.000 Y525.400 Z52.000 B180.000 C0.000 F250.000
N00330 M08
N00335 G02 Y474.600 B-180.000 I-0.000 J-25.400
N00340 X502.390 Y525.287 B84.600 I0.000 J25.400
N00345 G01 X500.000 Y525.400 B180.000
N00350 M09
N00355 M99
%
```

REFERENCES

Owen, Jean V. "Simulation: Art and Science." *Manufacturing Engineering* 114, (1995): 61-63.

Owens, John. "Robot Simulation - Seeing the Whole Picture." *Industrial Robot* 18.4 (Winter, 1991): 10-12.

Weisel, W. K. "Simulation Packages Change the Face of Robotics." *Robotics World* 14.1 (Spring, 1996): 40-41.

Roos, Eberhard and Arno Behrens. "Off-line Programming of Industrial Robots – Adaptation of Simulated User Programs to the Real Environment." *Computer in Industry* 33.1 (Summer, 1997): 139-150.

Boud, A.C, and S. J. Steiner. "New Method for Off-line Robot Programming: Applications and Limitations Using a Virtual Environment." *IEE Conference Publication* 435 (1997): 450-455.

Bernhardt, R., M. Schahn, and G. Schreck. "Knowledge Based Off-line Programming of Industrial Robots." *IFAC Proceedings Series* 10 (1999): 443-451.

Fujiuchi, Makoto, et al. "Development of a Robot Simulation and Off-line Programming System." *SAE Technical Paper* (Sep. 1992): 69-77.

Danni, L., et al. " Off-line Programming of Flexible Welding Manufacturing Cells." *Proceedings of the International Offshore and Polar Engineering Conference* (May 1996): 172-176.

Lee, D.M.A., and W.H. ElMaraghy. "ROBSIM a CAD-based Off-line Programming and Analysis System for Robotic Manipulators." *Computer-Aided Engineering Journal* 7.5 (Oct. 1990): 141-148.

Rooks, Brian. “Off-line Programming: A Success for the Automotive Industry.” *Industrial robot* 24.1, (Winter, 1997): 30-34.

Kamisetty, Krishnavrasad V., and Kevin J. McDermott. “Development of a CAD/CAM Robotic Translator for Programming the IBM 7535 SCARA Robot Off-line.” *Computers in Industry* 20.2 (Aug. 1992): 219-228.

Wozniak A., and J. Warczvnski. “Robot Simulation and Programming System.” *IFAC Proceedings* 10 (Oct. 1989): 437- 442.

Fukuda, Hideaki, Tsudoi Murakami, and Tateo Kojima. “Off-line Robot Programming System with Personal Computer.” *KOBELCO Technology Journal* 13 (Apr. 1992): 39-41.

Zeghloul, S., B. Blanchard, and M. Ayrault. “SMAR: a Robot Modeling and Simulation System.” *Robotica* 15.1 (Jan. 1997): 63-73.

Costagliola, G., et al. “Framework of Syntactic Models for the Implementation of Visual Languages.” *IEEE Symposium Proceedings on Visual Languages* (Sep. 1997): 58-65.

Rackovic, M. “Construction of a Translator for Robot Programming Languages.” *Journal of Intelligent and Robotic Systems* 15 (Feb. 1996): 209-232.

Levine, John, Tony Mason, and Doug Brown. *Lex & YACC*. Cambridge: O’Reilly & Associates, 1995.

Breuer, Peter, and Jonathan Bowen. “A PREttier Compiler-Compiler: Generating Higher Order Parsers in C.” *Software - Practice and Experience* 25.1 (Nov. 1995): 1263-1297.

Breuer, Peter, and Jonathan Bowen. “A PREttier Compiler-Compiler: Higher Order Programming in C.” Fifth International Conference on Software Engineering and its Applications. Toulouse, France, 7-11 Dec. 1992.

Breuer, Peter, and Jonathan Bowen. "The Cutting Edge of Parser Technology." Fifth International Conference on Software Engineering and its Applications. Toulouse, France, 7-11 Dec. 1992.

Breuer, Peter, and Jonathan Bowen. "The PRECC Compiler-Compiler." UKUUG/SUKUG Joint New Year 1993 Conference. Oxford, UK, 6-8 Jan. 1993.

Breuer, Peter, and Jonathan Bowen. "A Concrete Grammar for Z." Oxford University Computing Laboratory Technical Report. Oxford, UK, 15 Sep. 1995.

Banks, Jerry. *Handbook of Simulation* New York: John Wiley & Sons, 1998.

RRS Maintenance Management. *RRS Interface Specification Version 1.3.* Berlin: Frauenhoffer Press, (1997).

Flow Software Technologies. *Workspace 5.0 User Guide.* Newcastle Upon Tyne: Wex Tech Publishing, 1999.

Allen-Bradley Automation. *9 Series CNC Operation and Programming Manual.* A-B Press, 1997.

Breuer, Peter, and Jonathan Bowen "PRECCX User Manual." PRECC – A PREttier *Compiler - Compiler.* 11 January 1999 <http://archive.comlab.ox.ac.uk/redo/precc.html>.

Flow Robotics Company. Various G-code part programs, and machine specifications. Jan. 1999 – Sep. 2000.

Nof, Shimon. *Handbook of Industrial Robotics.* 2nd ed. New York: John Willey & Sons, 1999.

Willnow, Cornelius, et al. *Proceedings from RRS conferences.* 2nd ed. Berlin: Fraunhoffer Institute of Robotics, 1996.

Assarsson, Bo. "Robotized Waterjet Cutting." *Industrial Robot.* 21.1 (1994): 12-17.

Waterjet Web Reference. Home Page. 5 June 2000 <http://www.waterjets.org>.

OMAX Abrasive Waterjets – Precision computer-controlled abrasive waterjets, high-pressure pumps. Home Page. 6 June 2000 <http://www.omax.com>.

Flow International Corporation – waterjet cutting and cleaning, abrasive waterjet cutting system. Home Page. 10 January 2000 <http://www.flowcorp.com>.

Waterjet Technology. Home Page. 15 May 2000 <http://www.waterjet-tech.com>.

Banks, J., et al. *Discrete-Event System Simulation.* 2nd ed. New Jersey: Prentice Hall, 1996.

Law, A. M., and W. D. Kelton. *Simulation Modeling and Analysis.* 2nd ed. New York: McGraw-Hill, 1991.

Pegden, C.D., R. E. Shannon, and R. P. Sadowski. *Introduction to Simulation Using SIMAN.* 2nd ed. New York: McGraw-Hill, 1995.

Denavit, J., and R.S. Hartemberg. "A Kinematic Notation for Low-Pair Mechanisms Based on Matrices." *ASME Journal of Applied Mechanics.* (June 1965): 215-221.

VITA AUCTORIS

NAME: Aleksandar Z. Boskovic

PLACE OF BIRTH: Belgrade, Yugoslavia

YEAR OF BIRTH: 1970

EDUCATION: IV Beogradska Gimnazija, Belgrade 1985-1989

Belgrade University, Belgrade, Yugoslavia 1990-1997 B.Sc.

University of Windsor, Windsor, Ontario 1998-2000 M.A.Sc.