2001

Automatically generated object-oriented genetic programs to optimize adaptive job shop control and scheduling system.

Huining. Yuan
University of Windsor

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Automatically Generated Object-Oriented Genetic Programs to Optimize Adaptive Job Shop Control and Scheduling System

By

Huining Yuan

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science
at the University of Windsor

Windsor, Ontario, Canada
2001

Abstract

The requirements for Production Planning and Control (PP&C) System have fundamentally changed during last years. The increasingly complex production processes and constantly changing production environment require the system able to be agile, flexible and adaptable to the changing situations from markets, customers, new technology, environment, and so on. Fraunhofer Institute of Production and Automation (IPA), Stuttgart, Germany created an experimentation environment of software agent, event-oriented simulation and evolutionary strategies, to examine adaptive approach for the PP&C system. The project of Agent Learning Adaptive Network (ALAN), Fraunhofer – IPA proposed, focused on the Job Shop Control level to explore the new order management paradigm applicable to small and medium sized enterprises (SMEs).

Object-oriented programs automatically generated by Genetic Programming are expected to automatically co-ordinate between multiple intelligent agents to reach system's global targets. This research extends genetic programming beyond its current generation of functional and procedural programs to the generation of object-oriented programs. Successful achievement of this goal will represent a significant advance in the practice of genetic programming in Object-Orientation.

Keywords: Production Planning and Control, Job Shop Control, order management, optimization, Genetic Programming, evolution strategy, agent.

To my parents,
who taught me
"Survival of the fittest"
from real life.

Acknowledgements

I would like to express my sincere appreciation to Dr. Arunita Jaekel, for her supervision, for her consistent assistance and support during the whole course of my Master studies at University of Windsor. She gives me the most academic freedom as well as the most academic discipline. Her insight to co-operation education makes my educational philosophy and belief fulfilled.

I wish to acknowledge and thank Dr. Xiaojun Chen and Dr. Xiang Chen for serving on my committee, spending their time in reading my reports and giving me their instructive advice.

A special acknowledgement is due to Dipl.-Ing. Benno Loeffler, my supervisor and project leader in Fraunhofer – IPA, who spent countless hours, in explaining me the project architecture, correcting my understanding, discussing related issues in the project, so on and so forth. I appreciate the slides, the documents he made, especially for me, in English, and his jokes, even during the hard time of project.

I am grateful to the ALAN team, for the time, the ideas and the intelligence we have shared, and to other IPA employees, which helped me in picturing a group of excellent German engineers and researchers.

Words cannot adequately express my gratitude to Arslan Khan's help for my past two years' study. He pushed me through the hardest parts of the journey, and kept his belief in me and my achievement.

Table of Contents

List of Figures

Chapter 1
Introduction

On 21st June 1999, a memorandum of agreement was signed between University of Windsor and Fraunhofer-Institut für Produktionstechnik und Automatisierung (IPA) of the City of Stuttgart, Germany.

As per this agreement, the two parties agreed to initiate a Pilot Graduate Internship Program in Computer Science and Engineering, thus allowing students registered in an appropriate graduate program at University of Windsor to be engaged by IPA as interns working on projects in Germany and also that they will receive credit toward the graduate degree for research conducted in the course of the cooperative internship. A group of 3 students were selected by a panel of IPA engineers. I was among those fortunate ones.

The Fraunhofer-Gesellschaft is the leading organization of applied research in Germany. It currently operates 48 research institutes at 38 locations throughout Germany, with about 9,600 employees, most of them are scientists and engineers. The annual research budget amounts to over 1.5 billion German marks since 1999. FhG-IPA (Fraunhofer-Institute for Manufacturing Engineering and Automation) is one of these 48 Fraunhofer-Institutes. IPA is providing a flow of know-how and technology from the researcher in the university environments to industrial enterprises.

I joined IPA in January 2000 and was immediately assigned to the project of Adaptive Learning Agent Network (ALAN) to work with the ALAN team. One main focus of the project is to improve the adaptability of Production Planning and Control (PP&C) system to the constantly changing situations in production processes, so that a self-adaptive PP&C software system is about to be offered to the small and medium sized enterprises (SMEs) to optimize their production.

The fundamental innovation of ALAN is to advise the agents a global system target and automatically adapt each agent's behavior to reach such target. The new order management methodology is applied in the system implemented by the up-to-date IT technology, such as machine learning, intelligent multi-agents, genetic algorithm and genetic programming.

My main task was to implement the object-oriented Genetic Programming. Based on GP evolutionary strategies, agents can start to corporate with each other and work out a dynamically adaptive rule to direct the manufacturing system. The time I was in ALAN was the first phase of the project – ALAN prototype. The work focused on the definition of agent-model structures, basic control loops as well as business and enterprise scenarios. This thesis will give an overall view of the ALAN project and will cover some important aspects of Genetic Programming.

The second chapter mainly defines the research problems, analyzes the existing Production Planning and Control systems and proposes our new approach and methodology. Since this work is based on Job Shop Control level, the problem definition of Job Shop Control is given in this chapter, as well.

The third chapter overviews ALAN project. The mechanism of dynamic permanent adaptation applied in ALAN project is discussed; simulation, evolution and agent technology is overviewed; ALAN modules are described in detail.

As background knowledge, Chapter 4 gives introduction to the Genetic Algorithm and Genetic Programming and Chapter 5 discusses the GP extension, such as Koza's (1992) Automatically Defined Function (ADF), Montana's (1994) Strongly-Typed GP and issues related to GP object-oriented programming.

Chapter 6 delves into ALAN GP design, which is the main focus of my work in ALAN. The following aspects are discussed in detail: ALAN GP implementation language – Interpreter, program representation, population initialization, GP opertations, GP control and fitness evaluation. The implementation of object-oriented Genetic Programming is one of my main contributions in ALAN, and hereby described and analyzed in detail.

Chapter 7 analyzes the ALAN experiment results, disscusses some related questions, problems and future work. Chapter 8 gives some concluding remarks.

Chapter 2 Next-Generation Production Planning and Control System

2.1 What is Production Planning and Control (PP&C)?

Production planning and control is a production process of developing and using information to forecast market requirements, to efficiently manage the material flow, to coordinate internal and supplier activities, and to effectively use resources. The production planning and control represents a fundamental part of modern manufacturing technology. There are many existing PP&C mothods and tools, according to different levels of planning and control.

2.2 New Requirements

The requirements for PP&C system have fundamentally changed in recent years. With the increasingly complex production systems and spreading of flexible manufacturing, today's competitive global marketplace requires the manufacturing system:

1. to be able to produce various products from various limited and shared resources;

2. to be able to adapt quickly to changed conditions, e.g. markets, customers, products, new technologies, as well as production environments, including machine breakdown and product yield loss;
3. to be able to support optimally the achievement of the enterprise goals which are not constant all the time during manufacturing;
4. to be able to have orders commissioned "just in time", because the customers are not willing to accept long lead times;
5. to be scaleable and have the ability to integrate such systems quickly in the enterprise;
6. to be able to increase manufacturing productivity and quality.

These requirements change the way that the traditional production planning as well as order management used to be done. Production processes in increasingly complex manufacturing systems have to react quickly and adapt themselves dynamically to the market situation, customer requests, production environment and other unpredictable influences. The system agility, flexibility and adaptability has already become the key to enterprise success.

Production planning and control is no longer a static task. The production enterprises live in an environment that is changing rapidly, constantly and unpredictably. The permanent modification of supply and demand (Westkämper, Balve & Wiendahl, 1998) requires that PP&C system evolve to meet changing environments.

2.3 Existing PP&C systems

The flexibility of modern production planning has reached certain extent that the traditional control procedures can rarely exploit the potential of the devices.

Today's systems can't fulfil the above-mentioned requirements.

- Today's conventional planning system is based on detailed long-term forecasts. Disciplines like the chaos theory illustrates that minor disturbances cause disproportionate effects (Bergé ,1984). A number of unpredictable input factors, such as of resource failure, delivery bottlenecks, etc., have an impact on the enterprise behaviour and make a detailed long-term planning process fail. Moreover, much more frequent adjustment to production process is hardly managed by long-term planning system.

- Existing systems handle the unpredicatable changes and system flexibility by excuting very expensive replanning cycle repeatedly, and therefore, cementing the workflow.

- The enterprise workflow and communication are limited to a stationary status. Any unprecitable change can not be reported and handled promptly. Therefore, a decentralized enterprise supply chain network is demanded.

- The hierarchical planning frameworks, which are widely used in today's systems, do not reflect the existing enterprise organisation. Years of experiences show that these frameworks are difficult to handle and do not provide optimal results.

The potentials of new enterprise structures are taken insufficiently by existing production planning methods and systems. It is observed that the conventional procedures for planning and scheduling are not any longer suitable for the new enterprise paradigms. New concepts and strategies for the next generation enterprises are about to be exploited.

2.4 New Approach

Modern planning and control tools meeting present requirements are no longer conceivable without dynamic adaptability. Self-adaptation will, and must be one of the critical features, presented in the next generation PP&C system and take a continuously growing advantage in competition.

2.4.1 Objective

The objective of the project is to improve the adaptation, automation, standardization, scalability and customizability of system to enable Small-Medium Enterprises (SMEs) to

react quickly to the dynamics of their markets and cope with fast changes in the environment.

In this research, we desire a PP&C system which has the ability to:

- Improve responsiveness to customers[1], and be customer-oriented;
- adapt its structure according to new technology;
- fulfill "manufacturing on demand" principle and be fully flexible to market;
- provide high flexibility and rapid adaptability to potential perturbance variables;
- short-term forecast for unanticipated occurrences;
- meet high planning reliability by improving synchronization of internal and distributed planning.

2.4.2 ALAN Approach and Methodology

An "automatically and permanently adaptive" PP&C system was proposed, as Adaptive Learning Agent Network (ALAN) to achieve the objectives, mentioned as above.

ALAN automates enterprise transactions using adaptive agents and by connecting these agents to a logistic information infrastructure. Agents generate dynamic, self-adaptive

[1] Customers might be another producer, a wholesaler or a retailer, or customers alike.

planning rules by evolutionary strategy and use "sensor" to trigger the rule modification when any unanticipated change occurs in the system.

There has already been some approaches to accomplish environment modifications: concepts and paradigms for permanent adjustment are e.g. 'Die fraktale Fabrik' (Warnecke, 1993), 'Bionic Manufacturing' (Bongaerts, Wyns, Detand &Van Brussel, 1996), 'Genetic Manufacturing' (Ueda, 1993), 'Random Factory' (Iwata & Onosato, 1994), 'Virtual Manufacturing' (Kimura, 1993) and 'Holonic Manufacturing' (Bruns, 1996). Many of these approaches show that the optimization of an enterprise has to be considered comprehensively, in order to achieve significant improvements.

The approach presented here combines known planning and optimization methods into an automatically and permanently optimizing, adaptive controller consisting of controller elements that can be modeled on the shop floor control level. In the controller, each manufacturing element is represented by a software agent, which is intelligent, adaptive, and capable of coordinating with other agents by evolution strategy.

All the technologies like genetic algorithms, genetic programming, discrete event simulation and agent technology have been in use for several years. The new way of combining them, as presented in ALAN, will result in a very powerful way of implementing adaptive systems for production planning and control.

2.5 Job Shop Control

There are different levels of production planning and control. ALAN is based on Job Shop Control level, discovering the optimization rules between jobs and machines.

2.5.1 Problem Formulation

The Job Shop Control Problem (JSCP) is among the toughest real world combinatorial optimization problems. It was shown to belong to the NP-hard class (Taillard, 1994).

Basically JSCP is formulated as a set of **J** jobs that are processed on a set of **M** machines. Each job j ($1<=j<=n = |\mathbf{J}|$) consists of m operations (O_{j1}, O_{j2}, ..., O_{jm}) that must be scheduled in a pre-defined machine sequence imposed by technological requirements. Each operation is processed on each machine only once and without preemption. No operation may free its machine until it is finished. And each machine can only process one operation at a time. There is also a specified processing time $t(j,k)$ for each operation O_{jk} (job j in machine k). The problem is to find a sequence of jobs for each machine satisfying the technological restrictions and such that the finishing time of the last operation (completion time) on the latest job be minimized (Makespan minimization). The problem of large production processes is very difficult to solve due to the high combinatorial complexity resulting from their flexibility in processing multiple products.

2.5.2 Recent Research

Many procedures have been proposed to solve general job shop problem. The algorithms differ in methodology, solution quality and efficiency. Branch and Bounds method provides the optimal solution for this problem but its computation becomes prohibitive for large size instances. The most popular algorithms nowadays are iterative optimization algorithms, also called neighborhood search. Examples of advanced local search strategies are simulated annealing, taboo search, variable-depth search, neural networks and genetic/evolutionary algorithm. Local research approaches like those analyzed in (Aarts et al., 1997) and (Vaessens et al. 1996) can get good quality solutions without ensuring optimality.

There have been a number of studies on applying GA to JSCP (see (Davis, 1985), (Yamada and Nakano, 1995), (Kobayashi et al., 1995), (Yamada and Nakano, 1996), (Shi et al., 1997), (Yamada and Nakano, 1997), (Gen and Cheng, 1997)). Among the most successful ones, we can cite (Yamada and Nakano, 1995) and (Yamada and Nakano, 1996).

Chapter 3
Project Overview

In the project of Adaptive Learning Agent Network (ALAN), Fraunhofer Institute of Production and Automation (IPA) created an experimentation environment of software agent, event-oriented simulation and evolutionary strategies, to examine adaptive approach to the PP&C system. ALAN provides an electronic method for generating an adaptive solution in the manufacturing environment.

3.1 Agent-Based Controller – Solution of "Permanent Adaptation"

As discussed in Section 2.4.2, the core concept of ALAN approach is an automatically and permanently adaptive controller. The controller elements are adaptive software agents.

The system operates in two modes: A training mode precedes the operating mode. During the training mode, a model of the reality with all planning relevant items e.g. orders and jobs, is created. Each item is equipped with logistic key parameters pertinent to manufacturing, which determine the behaviour of the item in the model. Target of the "Meta Optimization" is finding suitable rules for all model items with consideration of an overall target system by evolution strategy. The quality of the newly found rules can be

measured on the degree of completion of the target parameters of the target system. During this mode, the training model serves as evaluator for the quality of the regulation model. As soon as suitable rules are found, the training mode is finished. In the following operating mode, commands are no longer sent to the training model, but to real manufacturing instead. The regulation model is now operated in real time, i.e. the execution is slowed down, so that the synchronization with real manufacturing processes is guaranteed. The manufacturing state's feedback now also comes from real manufacturing (see Figure 3-1).

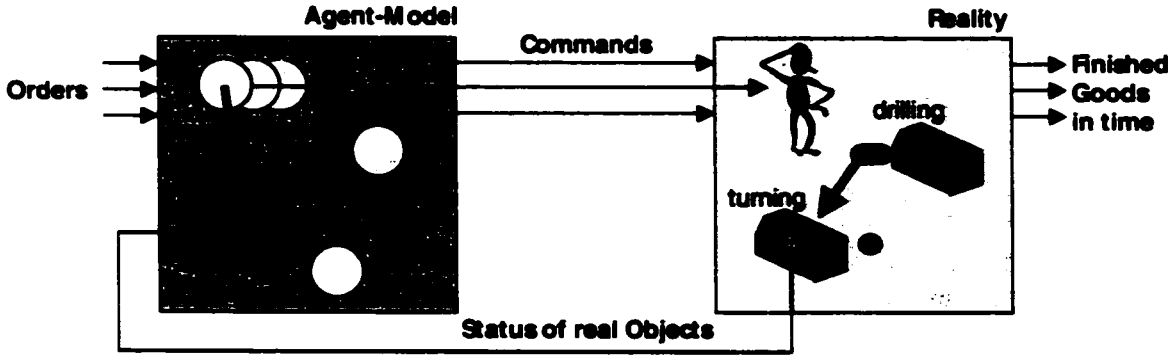

Figure 3-1: Agent-based Controller in Operation Mode

A manually configured discrete event simulation model calculates necessary actions in real time and the reality is controlled according to the actions happening in the model. Inevitable discrepancies between reality and model's state, caused by disturbances like fading of processing lead times, resource failures are immediately fed back into the controller model.

Thus, in combination with a current simulation model of manufacturing, a precise short-term forecast is generated. The better the controller model matches the reality, the better the regulation works in the real world manufacturing system. Manufacturing's behavior is observed in short control cycles. Since the controller is an event-oriented system, any necessary reactions are triggered. The controller thus learns goal-oriented counteracting in case the forecasted development takes an undesired turn. Such a system no longer works like conventional order management systems, but rather as an "agent-based controller".

3.2 Agents

In ALAN, the controller is built as an object-oriented model of planning relevant elements. These elements are represented by software agents. In contrast to the modeled, real objects, each agent has inputs and outputs and shows certain behaviors. The controller's global algorithm is based on the agents' behavior and cooperation. An agent's behavior can be depicted and executed through rules, neural nets, or special symbolic program languages. Inputs and outputs are used for the communication between the agents. Machines, orders, and lots are represented by agents.

This object-oriented modeling and depiction of the controller provides the user with new possibilities to have an impact on planning and control processes. Furthermore, the

software agent can take on monitoring functions for the element of reality assigned to him. In this way, an object-oriented monitoring system is implemented.

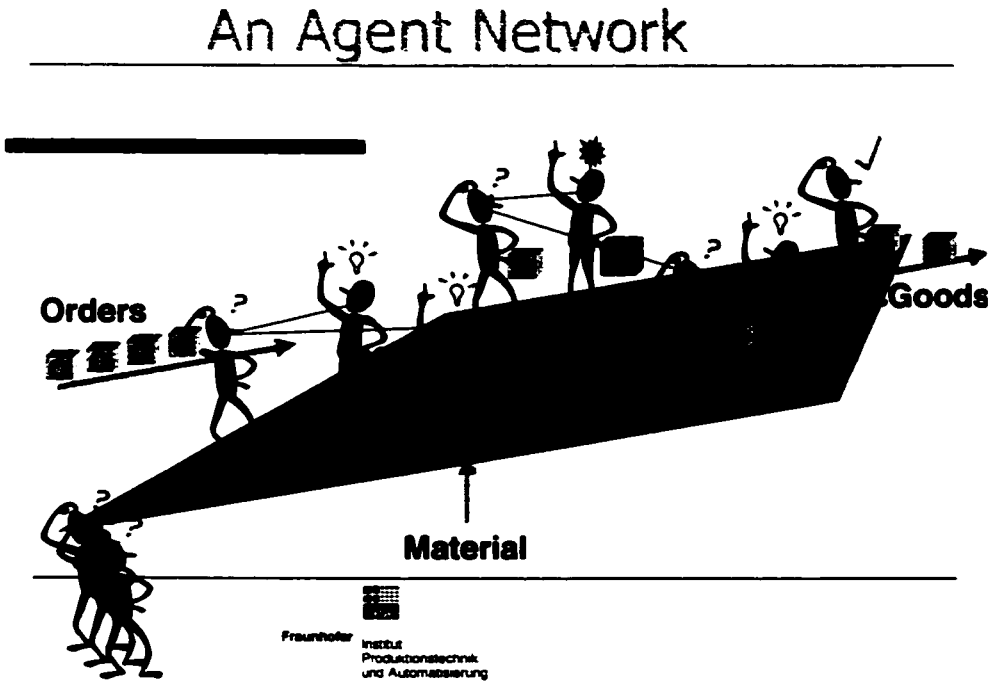

Figure 3-2: An Agent Network

For the training of the controller, an evolution strategy is used to manipulate the agents' behaviour, to evaluate the fitness of the collaborating agents, and therefore, the overall fitness of the controller, which consists of several agents. In this way, the global behaviour of the controller, e.g. trying to meet all due dates while having low stock level, is optimized. During the training mode, a simulation model is needed. The controller is trained to cope with the simulation model objects. During the operation mode, the controller interacts with real-world objects. The user can manipulate the control

parameters at any time and the agents can provide the system with additional control information.

3.3 Simulation

Planning and optimization of production and logistic systems touch sensitive areas of the enterprises. Analytical methods are rarely successful in the solution search in complex systems. The use of systematically or stochastically run "attempts", e.g. in the form of simulation methods and evolution strategies, have proven successful for problems of this kind (Becker, 1991).

Simulation opens completely new perspectives in practice. Expressive simulations are especially suitable for complex tasks and they are so low-cost that even small and medium-sized companies can profit from their advantages. Object-oriented simulation gives the strength of intuitive clarity of the problem's modeling.

Simulation methods copy reality in an executable computer model. By means of this model, simulation experiments about the exact effect of planning, operation and optimization measures can be carried out before it is executed. With this acquired knowledge from the future, risks, costs and the lead time of projects can be reduced to minimum.

Simulation makes it possible to test any changes in structure and parameters without risks of implementation or costs. The manual comparison of various simulation experiment results can be considered an optimization.

On the basis of the computer model, aim-oriented changes can be carried out and their consequences can be examined step by step within the actual simulation. It delivers detailed results, which – carefully interpreted and most exactly evaluated – form a solid foundation for decisions.

Therefore, during planning, new chances are opened up for optimization without any risk and cost - quickly, flexibly and realistically. The computer assisted simulation gives enterprise a clear competitive advantage – making decisions on the basis of realistic key figures – quickly and above all correctly.

3.4 Evolution

Evolution strategies go one step further. Evolution strategies stand out because of the problem independent automatism of stochastic solution search that follows a biological ideal.

Planning is the intellectual, goal-oriented anticipation of measures to be taken in the future. Each planning effort bears risks due to potential perturbance variables. Depending

on the networking of the plan elements and its overall fragility, an unexpected effect can necessitate partial or total replanning. The same holds true if a plan is generated through costly optimization methods.

The repeated manual improvement of the planning algorithm, as well as drawing up a machine plan, can in itself be understood as optimization. Instead of manual, analytical continuing development of the planning algorithm, an automatic evolution strategy can be used. The evolution algorithm can be executed by means of a simulation model of manufacturing, so that the risk of non-validated use of optimized algorithms in manufacturing is decreased.

3.4.1 Evolution Strategy

The parameter variations necessary to optimize a problem are carried out according to the biological evolution. Each potential solution, represented through a list of parameters, is considered as an individual. A number of solutions, i.e. individuals, make a population. All individuals of a population are evaluated according to their ability to solve the given problem. Depending on the evaluation, the individuals are introduced into a replication mechanism, which transforms an existing population to a succeeding population. The mechanisms used are reproduction, mutation, and crossover, as biology operations. These mechanisms, operating based on evaluation, therefore, improves individuals in following populations. In order to speed up the evolution processes and shorten optimization time,

evaluation has to be completed quickly and, if possible, automated. Parallel evaluation is suggested and implemented in ALAN. The described process, i.e. the evolution, is repeated until a break off criterion is fulfilled.

3.4.2 Method of Evaluation - Simulation

The new optimization rules, automatically determined, cannot be used immediately, before they are verified. Also, evolution strategies depend on the evaluations of solutions by an outside judge who works based on a goal system. Discrete event simulation performs this task. When manufacturing is mapped in an executable simulation model, optimization rules can be tested and evaluated through simulation. Corresponding to conventional optimization problems of logistics, business processes and resources, a model of the problem area is generated and validated. This model is then processed using different scenarios which are to be evaluated. Each simulation run provides a relative statement or value concerning the quality of the tested planning algorithm.

3.5 ALAN Architecture

ALAN is an object-oriented software framework for constructing Evolutionary Computing (EC) applications with extensible and reusable components. The following (Figure 3-3) is a list of ALAN Packages:

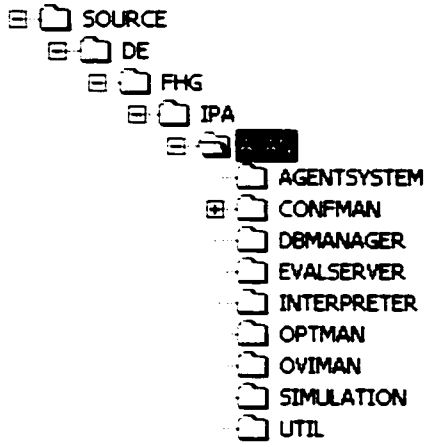

Figure 3-3: ALAN Components

ALAN applies the principles of design patterns and object orientation to construct a framework of extensible and reusable modules. The basic structure of the ALAN system is represented in figure 3-4.

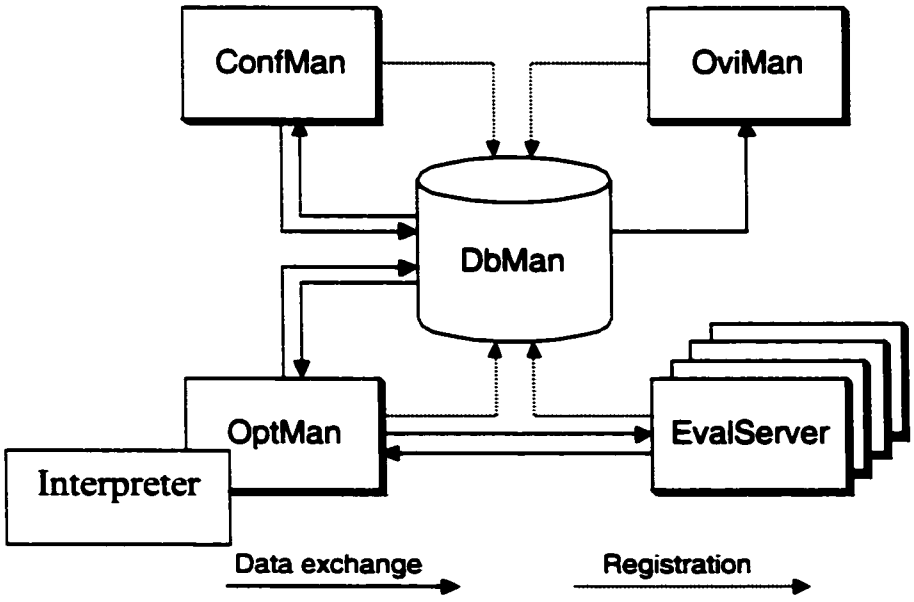

Figure 3-4: ALAN Modules

The central item - DbMan stands for Database Manager. It processes the largest part of the data exchange between the modules and administrates the resulting data. Basically DbMan manages the Database where all configurations, events, GA/GP runs are stored, such as production environments, products, scenarios, workplans, operations, resources, individuals with best fitness, evaluation servers, etc... The pure object database, „ObjectStore" PSEPro is applied to store the valuable data to make it persistent.

The ConfMan module, which stands for Configuration Manager, is responsible for all configuration functions. ConfMan contains a Graphical User Interface (GUI), which enables users to specify industrial scenario, production environment, GP parameters, etc.... The new configuration can be saved in the database and the previous configuration(s) can be retrieved from the database via the DbMan. These configurations can be defined or adjusted by users' specific needs.

OptMan - Optimization Manager, is responsible for the optimization of the program solution by conducting evolutionary computing. The OptMan retrieves the necessary parameters for the algorithms from the DbMan and sends the optimization results (the fittest program) back to DbMan. Since OptMan registers itself at the DbMan at the beginning of ALAN run, he is able to get the available EvalServer and send the newly generated individuals to have their fitness calculated there. Individuals are returned to the OptMan, after fitness assigned, ready for Genetic operations.

The EvalServer, Evaluation Server, receives individuals from the OptMan for discrete fitness calculation, as described above. The concept of parallel multi-EvalServer has the advantage to execute evaluation tasks on different computers in a network and therefore, make an optimal use of the available resources.

OviMan - Optimization Visualization Manager module offers an option to visualize the results of the optimization in the database, which is the individual stored in the database with the best fitness value. There are online and offline modes of OviMan. In the offline mode, the best solution so far will be transferred to the OviMan and be visualized; while in the online-mode, the best solution in each current generation will be displayed on the OviMan GUI. OviMan contains a Graphical User Interface with various visualization features.

The model of the production system is simulated on the basis of the start parameters defined by the user (manually configured in Configuration Manager). The results of this scenario are passed on to the evaluation module. Evaluation Manager distributes it to a remote machine to do simulation. The overall fitness value is calculated and is sent back to Optimization Manager for continuous improvement and optimization. Rounds of such process are implemented till the criterion is met. Thus, the solution, which met the goal of the system, is offered to the user.

3.6 ALAN Benefits

Such Production Planning and Control system improves business by:

- Increase of flexibility on the adaption to market changes
- Reduction of stocks in fractal structures (networks of business units)
- Customer orientation through the focus on customer demands and quality of deliveries (manufacturing on demand)
- Shortening process lead times in fractal structures
- Ability to handle price variation of the offered product
- Improvement of product quality
- Reduction of overall production and handling costs

The benefit of this new order management methodology is the use of the order management potentials in present and coming enterprise structures (e. g. fractal) and philosophies (e. g. manufacturing on demand). The high potential of scalability and adaptability exceeds those of traditional systems. Long-term planning is replaced by a multi-level control system with short-term forecasts. The concept of planning partially disappears and is replaced by short control loops. The new concept can be used in totally decentralized enterprise structures.

Chapter 4
Background Knowledge

This section attempts to provide some background knowledge and the mechanics of Genetic Algorithm and Genetic Programming.

4.1 Genetic Algorithm

Genetic Algorithm (GA) is a stochastic search technique which was introduced by John Holland (1975) at University of Michigan, Ann Arbor. The algorithm's search strategy is based on Darwin's biological evolution, using a computationally simulated version of survival of the fittest. It supports the view that the individual entities within a given generation that have the highest adaptability to the environment will be selected and survive into the next generation. This process thus generates entities that have obtained new characteristics or have received advantageous characteristics through genetic action.

Genetic Algorithm was inspired by the analogy of evolution and population genetics. GA model concepts from the evolutionary process found in nature to apply them to artificial optimisation problems. GA evolves solutions to difficult problems, where the answer is not obvious (NP problems). Genetic Algorithm will find a very good solution, but might not to be the best one.

4.1.1 Population Initialization

GA starts with an initial 'population' of trial solutions to a problem. The traditional simple GA, as defined by (Holland, 1975) represents the problem's data as chromosomes, each of which is a collection of genes. Holland's GA maps the genetic units into binary strings, where each bit corresponds to a single gene. A group of such binary strings is the population, with each individual of the population referring to a single possible solution to the particular problem.

4.1.2 GA operators

Having created a population, or first generation the next operation is the actual manipulation and application of the biological operators. As in nature, the biological operators do not work on the individuals as they appear in 'in real life' (phenotype), but on their chromosomal representation (genotype).

4.1.2.1 Reproduction Operator

Reproduction is the genetic algorithm analogy to the cloning operation in biological genetics. It is the cloning or copying of a selected individual into the next generation. The reproduction operator has the effect of maintaining useful genetic material in the population.

4.1.2.2 Crossover (Sexual Recombination) Operator

Crossover (Recombination), in essence, is simulated mating. There are many ways to define this operator but the basic principle is the same. Firstly a portion of the population is selected for crossover, according to certain mechanisms (details at 4.1.3 Selection). Individuals, which are selected, form a mating pool. Once in the mating pool, individuals are paired and the crossover process is carried out. Crossover has many definitions, the simplest of which is the following: a random position is selected along the length of the chromosome and the two individuals swap genetic information from this point onwards creating two new individuals. This is illustrated in Figure 4-1.

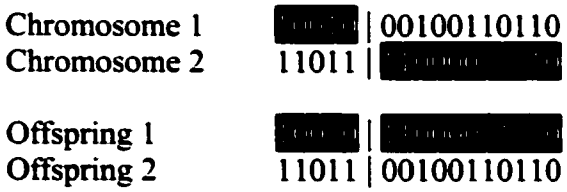

Figure 4-1: Crossover Operation in Genetic Algorithm (| is the crossover point)

4.1.2.3 Mutation (Asexual) Operator

Another common biological operator is mutation. This operator is intended to act as a means of introducing some random variation into the population. It is applied as follows: an individual is selected at random from the population according to a mutation probability. A position along the length of the chromosome is chosen at random and the

value of the gene in this position is flipped from 1 to 0 or 0 to 1. In this way, a small variation is introduced forming a new individual.

The mutation operator serves to maintain diversity in the population. Its use in conjunction with a large population of individuals helps the genetic algorithm to avoid local optima and convergence in the search space.

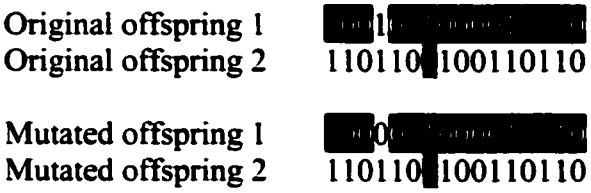

Figure 4-2: Mutation Operation in Genetic Algorithm (one point mutation)

Other forms of biological operators include inversion, permutation and editing all of which can substantially aid the genetic search and are described in (Goldberg, 1989), (Koza, 1992). A new population is bred after applying these genetic operators to the existing population.

4.1.3 Selection

The selection procedure is of vital importance to the likely success of the genetic search. The most common form of selection applied is referred to as **roulette wheel selection** (Goldberg, 1989). The principle behind roulette wheel selection is simply that an

individual with a higher fitness value should have higher probability to be selected for genetic operations.

Another population form of selection is **tournament selection** where two or more individuals are chosen at random to participate in a tournament. The fittest individual(s) will be the winner(s) and will be selected to participate in genetic operations.

Elitism is sometimes used in selection to preserve the best individual found in a run. This form of selection always passes a copy of the best individual of the population to the new generation.

Other selection methods include **Rank selection** and **Stochastic Universal Sampling**. A detailed analysis of each of them is beyond the scope of this work.

4.1.4 GA Fitness Evaluation

Selection is rated by fitness evaluation result. Fitness evaluation is conducted by fitness function. In Genetic Algorithm, fitness function is set on the high problem statement level and is the formula to decide how close a candidate solution actually is to solving the problem.

The nature of the fitness measure varies greatly from one type of program to the next. For some problems, the fitness of an individual can be measured by the error between the result produced by one candidate solution and the correct result. The closer this error is to zero, the better the candidate solution is.

4.1.5 GA Termination Strategy

By the use of genetic operators such as reproduction, crossover and mutation, a population of candidate solutions is continuously improved, through many generations, towards a not necessarily global optimum. The whole process repeats itself through successive generations, evolving new populations. Unlike natural evolution, which continues indefinitely, GA has to be stopped somewhere, when the termination criteria is met:

(1) solution is found, i.e. fitness is satisfied

(2) convergence occurs

(3) a certain number of cycles are completed.

Convergence is a phenomenon that all, or nearly all, individuals in the population have the same genotype or genetic representation.

4.1.6 GA Application

Genetic Algorithm has been demonstrated to be effective and robust in searching very large, varied, spaces in a wide range of application, including financial, imaging, VLSI circuit layout, gas pipeline control and production scheduling (Banzhaf, Nordin, Keller, & Francone, 1998).

4.2 Genetic Programming

4.2.1 What is Genetic Programming?

Genetic Programming (Cramer, 1985; Koza, 1992; Koza, 1994; Kinnear, 1994) is an extension of the genetic algorithm (Goldberg, 1989) in which the individuals in the population consist of computer programs represented as expression trees.

Genetic Programming is a domain-independent method for automatically creating a working computer program from a high-level problem statement that evolves computer programs that solve, or approximately solve, problems.

Genetic Programming is a relatively new and powerful approach to the automatic generation of computer programs from specifications of program behavior. Genetic programming combines ideas from the fields of biology (natural selection, survival of the fittest), genetics (reproduction, recombination and mutation of genetic material), artificial intelligence (state space search), and compiler theory (representation of programs as abstract syntax trees) to create an algorithm which searches a large but constrained space of computer programs that are capable of adapting or recreating themselves for one or more programs, which are nearly optimal in performing a specified open-ended task.

Genetic programming works best in complex problem domain, in which problems have no ideal solution. Furthermore, genetic programming is useful in finding solutions with a large number of variables, which are constantly changing and none of which is encompassing the measurement.

Genetic programming has been extensively applied to generate functional programs which solve difficult problems in application areas as diverse as automatic design, pattern recognition, symbolic regression, multi-dimensional least squares regression, music and picture generation, neural network architecture and training, robot behavior planning and many others.

4.2.2 GP versus GA

Genetic programming is a branch of genetic algorithms where the problem solution representation strings are replaced with variable length programs, instead of fixed-length binary strings.

Unlike Genetic Algorithms, the crossover points in both parents of GP programs do not have to be the same, and probably cannot be. It is likely that the parents will have a different number of nodes, and have a different configuration.

GP programs are evaluated for fitness by executing the programs. The programs, when executed, are the candidate solutions to the problem; while in GA, the fixed-length strings of parameters, themselves, encode possible solutions to a problem.

Genetic programming is much more powerful than genetic algorithms. The output of the Genetic Algorithm is a quantity, while the output of the Genetic Programming is another computer program. Many problems appear to be inappropriate candidates for solution via genetic algorithms because they, in effect, require the operation method (i.e. "how") that produces some desired output value when presented with particular inputs. In essence, this is the beginning of computer programs that program themselves.

Genetic Programming is descended rather directly from the GA paradigm. However, GP, as a field founded by John Koza at the beginning of the 1990s, has grown exponentially since then, and become a new branch of Evolutionary Computation and now is a separate, very successful branch of its parent field.

4.2.3 GP Significance

One of the central challenges of computer science is to get a computer to do what needs to be done, without telling it how to do it. Genetic programming addresses this challenge by providing a method for automatically creating a working computer program from a

high-level problem statement. Genetic programming achieves the goal of automatic programming (also sometimes called program synthesis or program induction).

Genetic Programming allows problems to be solved without explicitly programming the solution. It would be desirable if computers could solve problems without being explicitly programmed. Specifically, it would be desirable to have a problem-independent technique whose input is a high-level statement of the problem's requirements and whose output is a working computer program that does a satisfactory job of solving the problem. This kind of "What You Want Is What You Get" ("WYWIWYG" P pronounced "wow-eee-wig") method of creating a computer program is called automatic programming (Arthur Samuel, 1959). Automatic programming is one of the central problems of computer science.

Since the early 1990s, GP has emerged as one of the most promising paradigms for fast, productive software development. GP, as a method of developing software, is radically different from current software engineering practice. GP is a kind of heuristic weak search, in the sense that it does not use background knowledge intensively. (Aler, Borrajo & Isasi, 1998). Potentially, in GP, the domain experts, instead of trying to transfer their knowledge to computer programmers, can create programs by directly specifying how they should behave.

Genetic programming addresses the problem of automatic programming, namely the problem of how to enable a computer to do useful things without instructing it, step by step, on how to do it. The rapid growth of the field of genetic programming reflects the growing recognition that, after half a century of research in the fields of artificial intelligence, machine learning, adaptive systems, automated logic, expert systems, and neural networks, we may finally have a way to achieve automatic programming. Genetic programming is fundamentally different from other approaches in terms of (i) its representation (namely, programs), (ii) the role of knowledge (none), (iii) the role of logic (none), and (iv) its mechanism (gleaned from nature) for getting to a solution within the space of possible solutions. (Bruce, 1995)

4.2.4 GP Mechanism

GP consists of three main components: a population of individuals (or candidate solutions), a fitness measure (or heuristic function) and a set of genetic operators.

4.2.4.1 Creation of Initial Population of Computer Programs

A genetic programming run begins by creating an initial population (generation zero) of abstract syntax trees that have random shapes and sizes. The nodes in the program trees are functions and terminals appropriate for a given problem. The nodes in each initial tree are randomly chosen from the set of functions and terminals being used.

- The set of functions that appear at the internal points of a program tree may include ordinary arithmetic and logic functions, such as addition, subtraction, multiplication, division and other more complex functions; conditional operators, loops, etc... and user-specified functions.
- The set of terminals appearing at the external points (leaf nodes in program trees) typically include the program's external inputs (such as the independent variables x and y) and constants (such as 3.2 and 0.4).

Functions and Terminals are important components of genetic programming. They are the alphabet of the programs to be made.

4.2.4.2 Fitness Function

Fitness Evalution is one of the difficult and important concepts in Genetic Programming. The evolutionary process is driven by a fitness measure that evaluates how well each individual computer program in the population performs in its problem environment. In GP, the fitness of each individual in the population is assigned by executing that individual program with certain input values and measuring how well it performs at the problem one is trying to solve. The detail was discussed in GP versus GA.

4.2.4.3 GP Operators

Similar to Genetic Algorithm, three genetic operators are generally used in GP: reproduction, crossover and mutation. The genetic operators are applied to fitness proportionally. Therefore, genetic operators genetically search the space of possible computer programs for a fittest individual computer program.

Reproduction involves selecting a single abstract syntax tree, probabilistically selected based on fitness, from the population and making a copy of that tree.

In the **crossover**, two selected parents are usually of different sizes and shapes. A crossover point is randomly chosen in each of the two parents. Then, the subtrees rooted at the two crossover points are swapped and placed at the same position from which their counterpart was removed. This results in two new abstract syntax trees which contain genetic material from each of their parents, with typically high fitness. Crossover is the predominant operation in genetic programming (and genetic algorithm, as well) and is performed with a high probability (say, 85% to 90%). See Figure 4-4 for an example.

In Genetic Programming, identical parents can yield different offspring; while in genetic algorithms identical parents would yield identical offspring. This is once again the advantage of genetic programming over genetic algorithm.

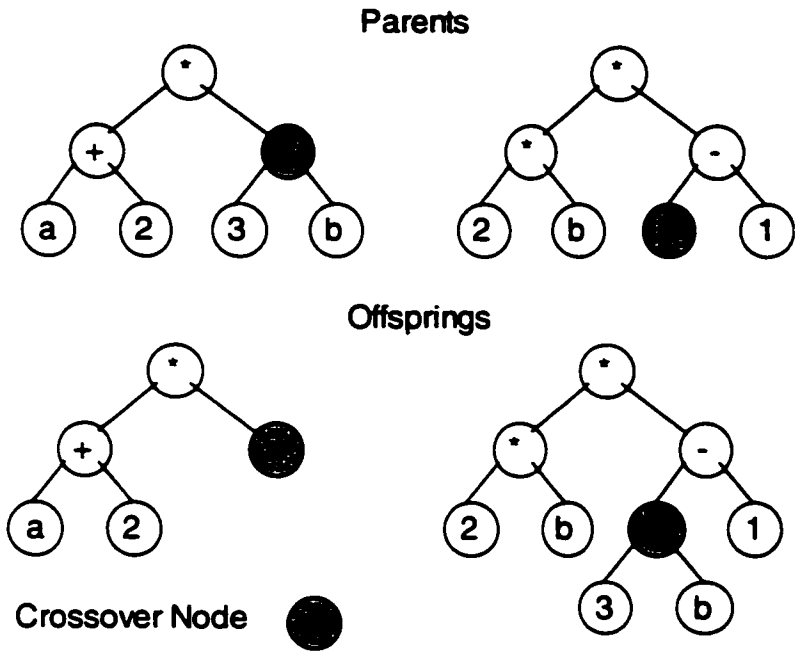

Figure 4-3: GP Crossover Example

Mutation is a process analogous to asexual reproduction in which a newly generated subtree replaces the subtree rooted at a random mutation point in a selected individual, probabilistically based on fitness. This asexual mutation operation is typically performed sparingly (with a low probability of, say, 1% during each generation of the run).

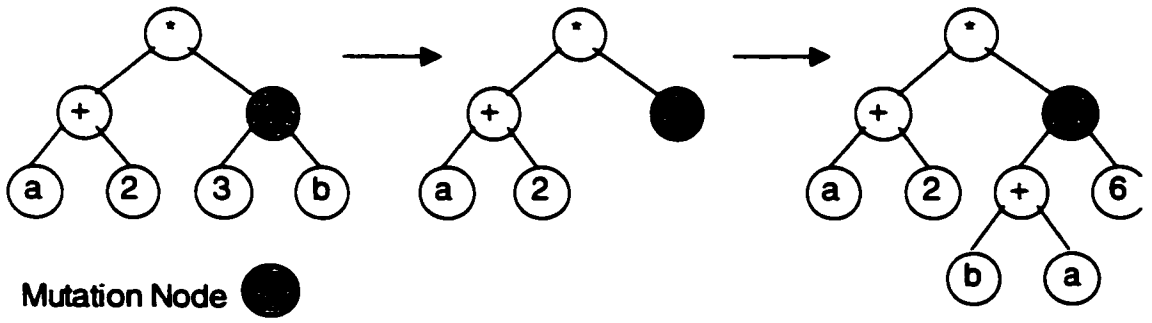

Figure 4-4: GP Mutation Example

4.2.4.4 Main Generational Loop of GP

Genetic programming breeds computer programs to solve problems by executing the following three steps:

1. Generate an initial population of random compositions of the functions and terminals of the problem (computer programs).

2. Iteratively perform the following sub steps until the termination criterion has been satisfied.

(a) Execute each program in the population and assign it a fitness value using the fitness measure.

(b) Create a new population of computer programs by applying the following two primary operations.

i) Copy the best existing programs;

ii) Create new computer programs by mutation;

iii) Create new computer programs by crossover(sexual reproduction).

3 The best computer program that appeared in any generation, the best-so-far solution, is designated as the result of genetic programming (Koza, 1992).

Flowchart for Genetic Programming

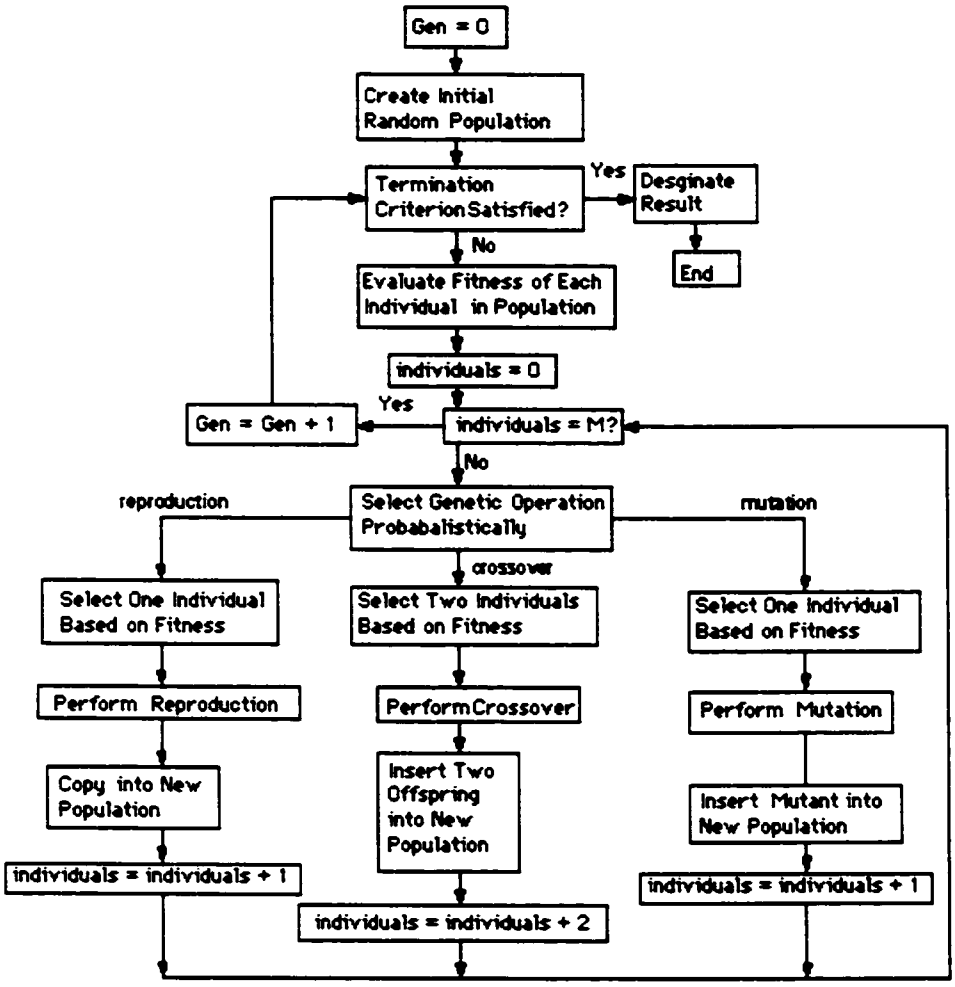

Figure 4-5: Genetic Programming Flowchart

4.2.4.5 GP Control Structure and Termination Criterion

The genetic operators and the policy of selection based on survival of the fittest provides genetic programming with a means to explore the search space. The specification of a

genetic programming system is completed with the choice of a search control structure and a search termination criterion.

The primary parameters for controlling a run are the population size and the maximum number of generations to be run. Secondary parameters include probability of reproduction, crossover and mutation, maximum depth of S-expression created in the initial population and after genetic operators, such as crossover and mutation, are operated.

The evolutionary process proceeds generation by generation. GP run will be terminated when it completes the number of designated cycles or when an individual with the perfect fitness is found. Usually the best-so-far individual is designated as the result.

The interaction of the genetic operators and the use of stochastic selection based on survival of the fittest results in an exploration of the search space which moves towards abstract syntax trees which have increased fitness on average over time.

Chapter 5
Genetic Programming Extension

This section presents several extensions to the genetic programming paradigm. These extensions attempt to improve the power of the paradigm by increasing the difficulty of the problems that can be solved by genetic programming and by extending the set of language features that can be used in programs generated by genetic programming. These extensions include Automatically Defined Functions (ADFs), Strongly-Typed Genetic Programming and Object-Oriented Genetic Programming.

5.1 Automatically Defined Functions (ADFs)

The standard genetic programming approach attempts to generate a single program in order to solve a problem. However, in many cases the best solutions for more difficult problems tend to be hierarchical in nature. The divide and conquer approach has been successfully used in human problem solving to deal with the intellectual complexity of these problems by dividing them into easier subproblems which can be directly solved and then reassembled into a global solution. Koza (1992; 1994; 1995; 1996) presents a method for using simulated evolution to automatically divide difficult problems into a main program and one or more subprograms whose solutions are simultaneously evolved

by a genetic programming system. This method is called Automatically Defined Functions (ADFs).

Simple computer programs consist of one main program (called a result-producing branch). However, more complicated programs contain subroutines (i.e. automatically defined functions, ADFs, also called function-defining branches), iterations (automatically defined iterations or ADIs), loops (automatically defined loops or ADLs), recursions (automatically defined recursions or ADRs), and memory of various dimensionality and size (automatically defined stores or ADSs).

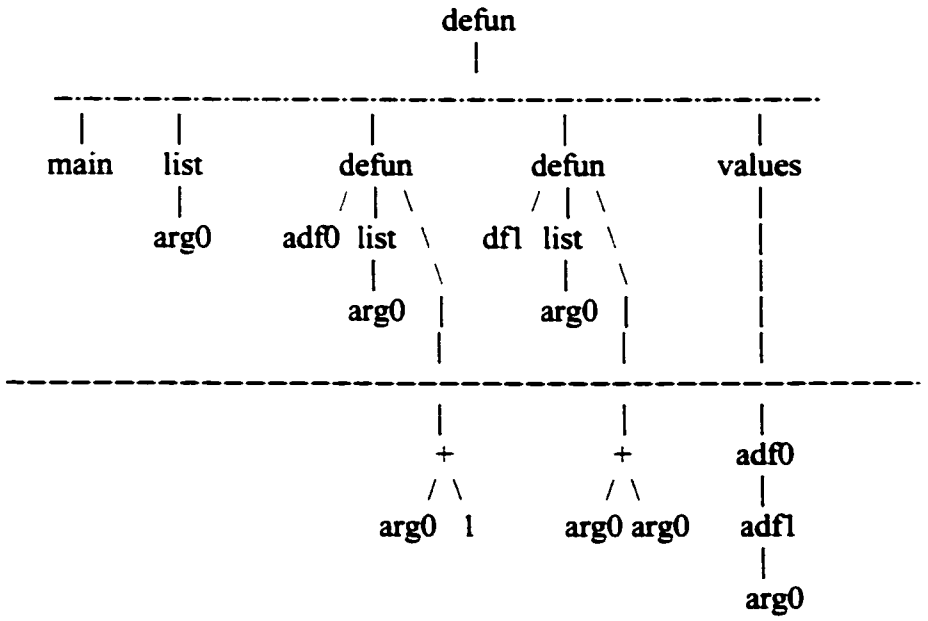

Figure 5-1: Structure of an Automatically Defined Functions

Figure 5-1 shows the tree structure of an automatically defined function. This figure is divided into two sections, those portions of the tree above the dashed line and those portions of the tree below the dashed line. The portions of the tree above the dashed line represent the fixed structure of the procedure that must be maintained in every individual in the population. The portions of the tree below the dashed line represent the parts of the procedure that can be evolved to solve the problem.

In this way, a problem can be solved by simply prespecifying a reasonable fixed architectural arrangement for all programs in the population (i.e., the number and types of branches and number of arguments that each branch possesses). Genetic programming can then be used to evolve the exact sequence of primitive work-performing steps in each branch.

Several sets of experiments have been performed to compare the performance of genetic programming systems that use automatically defined functions with those that do not (Koza, 1992 & 1994). The results of these experiments show that the use of automatically defined functions increases the power of a genetic programming system in terms of the difficulty of problems that it can solve. The technique also improves the speed at which solutions can be obtained for all but the most trivial of problems. In addition, the use of automatically defined functions generally results in more compact solutions than those found without the technique. The general reuse in the main routine of the subroutines evolved by the system allow for the improvement in program size.

The method of automatically defined functions is extended in (Koza, 1992) by allowing automatically defined functions to call other automatically defined functions. And several different architecture-altering operations, such as subroutine duplication operation, argument duplication operation, subroutine creation operation, subroutine deletion operation, argument deletion operation, etc. are each applied sparingly during the run, to reuse automatically defined function.

Genetic programming uses architecture-altering operations to automatically determine program architecture in a manner that parallels gene duplication in nature and the related operation of gene deletion in nature. Programs with architectures that are well-suited to the problem at hand will tend to grow and prosper in the competitive evolutionary process, while programs with inadequate architectures will tend to wither away under the relentless selective pressure of the problem's fitness measure. Thus, the architecture-altering operations relieve the human user of the task of pre-specifying program architecture. (Koza, 1999) Via ADFs and these related techniques, GP gains considerable benefit from functional abstraction.

5.2 Strongly Typed Genetic Programming

In the Koza approach (Koza, 1992), population individuals are defined as hierarchical (Lisp) Symbolic Expressions. S-expressions are composed of two types of functions,

functions that accept arguments and called non-terminal functions, (For brevity we use the term function in the previous chapters to mean non-terminal function) and functions take no arguments called terminals. No information on the data types of functions and terminals is used. Closure of the set of language element is required to be able to run the generated programs. That is, all functions have to accept arguments and return values of the same type. This allows for legal crossover recombination and mutation of S-expressions. This is called untyped genetic programming.

One important constraint of Koza GP is that S-expressions have to satisfy the property of closure. Montana has introduced Strongly-Typed Genetic Programming (STGP) by extending the GP paradigm to overcome the problem of closure. Montana (1994) uses the type information associated with variables, constants and the signature of functions to constrain the programs that can be generated by the initialization process and the genetic operators, so that only syntactic correct programs with regards to both structure and type can be constructed. Functions in STGP now can accept arguments and return values of different data types. Variables, constants, arguments, and returned values in GP can be of any data type with the provision specified beforehand.

Closure is not required in Montana's approach since syntactically invalid programs cannot be generated by the system.

The algorithm requires several modifications to include the use of type information. First, the random creation of programs must be modified so that when a node is randomly selected, the only nodes that are eligible for selection are those which satisfy the required return type. Similarly, the crossover and mutation operators must also be modified so that they only return syntactically correct programs with respect to the data types. In the case of mutation, the new node must return the same data type as the old node. In the case of crossover, the root node of the two subtrees to be swapped must have the same data type.

These modifications allow the generated abstract syntax trees to be translated into typed languages. They also constrain the search space during the program generation process by eliminating the need for the genetic programming system to induce the language type-syntax structure along with solving the actual programming problem.

5.3 GP Object-Oriented Programming

Genetic programming is a powerful new method for the generation of computer programs from examples of the program's desired input-output behavior. It is based upon the solid foundation of the field of genetic algorithms and provides the robust behavior of that paradigm applied to a new problem representation, abstract syntax trees.

One interesting area where genetic programming has not yet been widely applied is in the generation of object-oriented programs (Bruce, 1995). Since object-orientation is

becoming one of the more heavily used methodologies for constructing systems, it is of critical importance to the acceptance of genetic programming that it be extended to work in this area.

With the movement of industry towards the object model as the basis for analysis, design and programming, the application of genetic programming as a means for automating the programming of objects has the potential for positively impacting the software productivity crisis. Winter, McIlroy and Fernandez-Villacanas (1994) hint at this potential by projecting that the financial cost of using genetic programming to generate small program components will become less than the cost of using human programmers to perform the same task as early as the first decade of the twenty-first century. If these projections are correct, the use of genetic programming to build object oriented programs should be economically advantageous to organizations that wish to take advantage of the technology.

Chapter 6
ALAN GP Design

ALAN provides a collection of about 250 Java classes representing over 30,000 lines of code. As for the actual implementation of the genetic program, programming language Java is used, instead of LISP, for the reasons illustrated in the session of "6.1 Implementation Language Issues". The JDK (Java Development Toolkit) Version 1.3 is used to code the system.

6.1 Implementation Language Issues

There have been few experiments of GP systems implemented in Java (Chong, F.S. & Langdon, W.B., 1999) (Zalzala, A.M.S. and Green, D., 1999), but considering the relative youth of both GP and Java, it is not too surprising that Java has not been applied to GP with much frequency. ALAN was implemented in Java, for the following main concerns:

1. The Java Virtual Machine (JVM) provides platform independence and full portability, allowing ALAN to be used on most major platforms. The JVM's support for automatic memory management also allows for straightforward implementation of tree sharing which keeps memory usage small even with large populations.

2. Java is the recognized computer language for client server distributed and multi-threading parallel applications.

Basically, Genetic Programming requires substantially higher computing power and memory. GP is inherently a computational intensive process due to the many evaluations and manipulations of a potentially large number of programs, over a large number of training cases. A way of reducing the computational load, and hence increasing the computing speed and power, is to parallelise the evaluation process and use a multi-processor system to evaluate sub-population.

Java fulfils the distributed approach for parallelizing Genetic Programming. A centralized controller assigns sub-populations of individuals to different computers, via Java RMI, in ALAN System, which would carry out evaluation within the JVM of each computer and then report the results to the centralized EvalServer. In this manner, huge populations of individuals are speedily evaluated in a massively parallel process.

3. We used a GP kernel written in Java, due to its good connection with pure Java object-oriented database – ObjectStore PSEPro.

ObjectStore Personal Storage Edition (PSE) Pro for Java (products of Object Design Inc) is a lightweight single-user 100% pure Java object persistent storage engine for Java

developers. PSE and PSE Pro for Java are pure object databases for embedded database and mobile computing applications. ObjectStore PSE/PSE Pro combines the simplicity of object serialization, and the reliability of a database management system (DBMS), together with in-memory-like performance for accessing persistent objects.

6.2 GP Object-Oriented Programming

In ALAN, Genetic Programming is extended to the discovery of multi-agent behaviors. Each agent is active, goal-oriented, responsible for a task. Each agent has its "hidden agenda" (internal behavior), sensors and actors to interact with environment (communication), and each of them has a „Real-World" object. In ALAN system, each order has an OrderAgent; each machine has a ResourceAgent to represent them.

All intelligent agents together act as a controller in ALAN, taking requirements from customers and giving command to the manufacturing system. The role of the agent will be that of using its sensors to gather information in they system, and its actors to act on it, while genetic programs generates commands for the actors based on sensorial information available and its current internal state.

The behavior of the adaptive agents is represented as a certain number of programs and modeled as functional blocks of executable code (by means of an interpreter). The optimization is manipulated by evolution of these programs. After optimization, those

agents behavior (the arrangement of the different functional blocks of programs) are rearranged to obtain new, improved (or nearly improved) behavior. The improvement can be measured with the help of simulation.

Since the purpose of this work is to discover the self-adaptive agent behavior, the GP work mainly focuses on how agents communicate and interact with each other. Therefore, this work extends genetic programming to the automatic definition and evolution of object-oriented programs, with order agents and resource agents as the main concern of the project.

Object-oriented GP instead of GP was used, because expressing the structure of ALAN system involves more than one type of classes. To disable crossover between types which are not related such as combination of OrderAgent with ResourceAgent will certainly optimize search space.

6.3 Program Representation (Interpreter Language)

An executable, object-oriented language was illustrated which allows changing the behavior of the agents at run-time of the system. ALAN's own Interpreter language was developed, its kernel written in Java.

This interpreter language is based on interpreter building blocks, represents a program with a parse tree structure and interprets each node of the tree at run-time of the program, so that an executable program is developed. In GP run, the program structure is about to be changed by the exchange, modification of the building blocks.

Program trees are usually represented top-down and read from above. Branches are processed from left to right.

First a simple function is to be represented as a function tree.

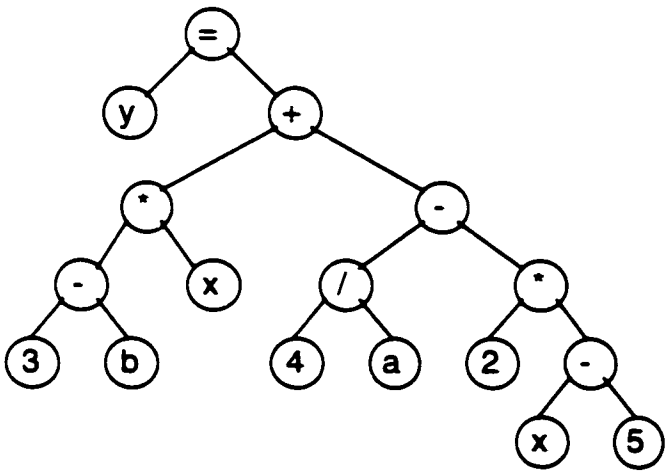

Figure 6-1: Function Tree

Figure 6-1 is the function tree that the function y = (3 - b) * x + (4 / a – 2 * (x - 5)) represents. However, except for arithmetic operations, other program items exist, such as conditional statement, loops, iterations, also need to be represented in the program tree.

Figure 6-2 shows a short program example with a IfThenElse statement and two Assignments.

```
x = 10;
if (x > 3){
  y = 1;
}
else {
  y = 0;
}
x = y;
```

Figure 6-2: Program Example

This program can be expressed as a program tree, like in figure 6-3 represented. Sequence, as the highest program node, contains three sub-programs - two assignments and a IfThenElse statement. In the assignments, either a literal value or another local variable value is assigned to a local variable in each case. The IfThenElse statement possesses a condition, with this condition fulfilled, the if-branch is executed, otherwise the else-branch is executed. Both branches contain an assignment once again.

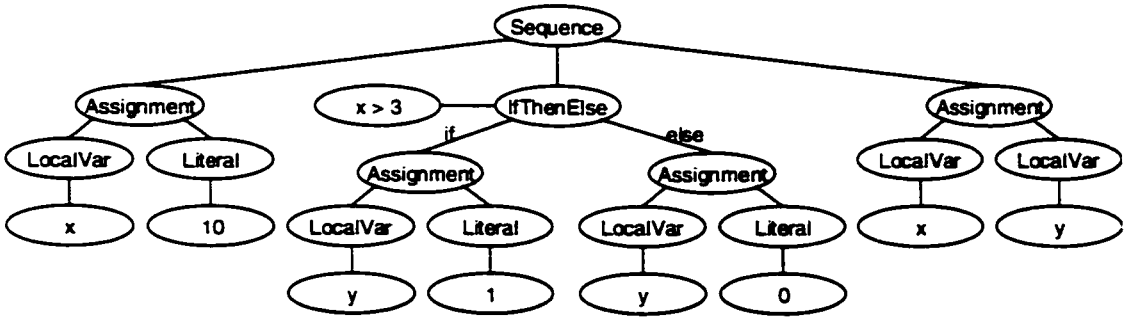

Figure 6-3: Program Tree of the Program Example

Now the interpreter building blocks represent a program showed as Figure 6-2, as a program tree. In our object-oriented Genetic Programming, each program node is an object and the class hierarchy of the individual interpreter building blocks is to be represented.

The abstract class Interpretable is the highest class of the class hierarchy of all classes, such as FunctionCall, Term and Expression. That means all these classes are inheriting Interpretable executable building blocks represent at interpreted run-time.

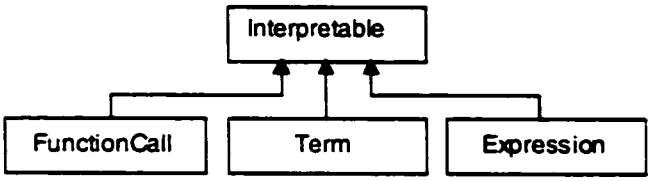

Figure 6-4: Abstract Class of Interpretable

Terms, such as sequence, IfThenElse, Loop, Assignment, VarDeclaration, are mainly the operation statements (programming operations, without return value required).

Expressions, such as Literal, LocalVar, DotOp and Return need to return values.

FunctionCall includes all the functions, such as standard arithmetic and logical functions – JavaCall (for basicInterpreterTypes, IInteger, IDouble, Ifloat and IBoolean) or domain-specific function – InterpreterCall (for behaviorAgent).

The function sets of trees are implemented by objects of the inherited classes of Term, Expression and FunctionCall.

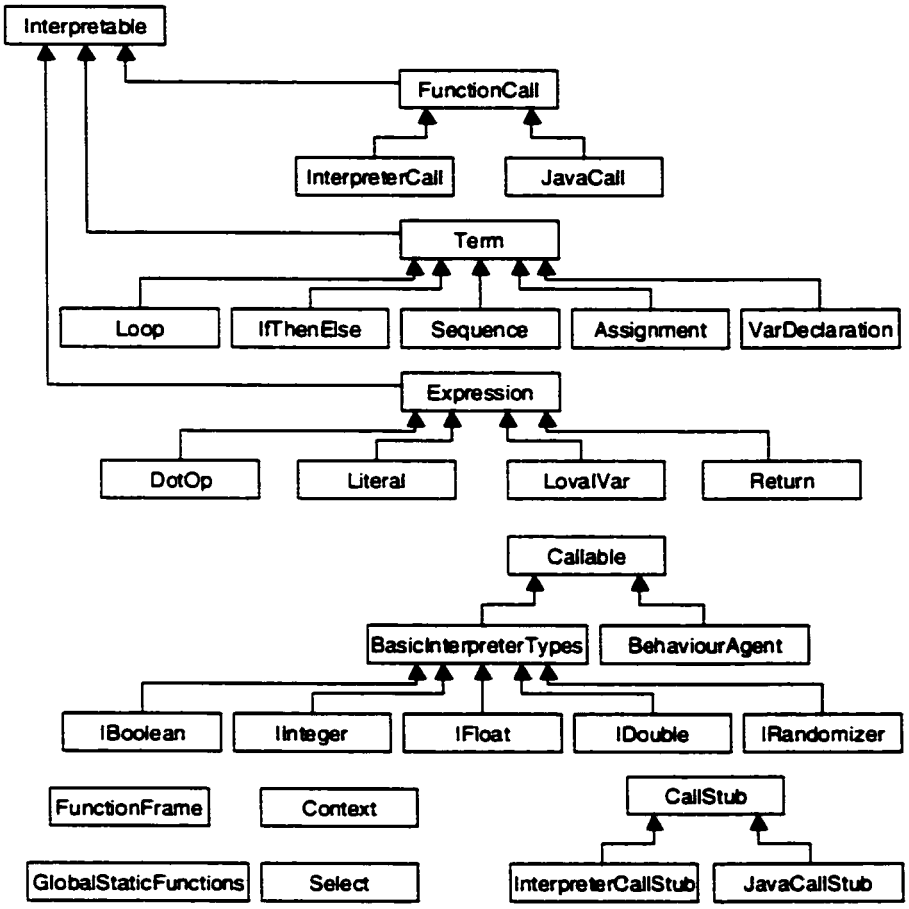

Figure 6-5: Class Hierarchy of the Interpreter Building Blocks

The interface Callable is implemented in all the cases that a specific class type is required. The abstract class BasicInterpreterTypes, BehaviourAgent and ManufacturingServices implement the interface Callable.

The abstract class of BasicInterpreterTypes provides the basic types that most present programming languages provide - Boolean, Integer, Float, Double. The class IRandomizer is to produce any of these types randomly, decided by the system.

BehaviourAgents includes OrderAgents and ResourceAgents. The domain specific functions – human coded functions and genetically generated functions will be applied to discover the best solutions to the Manufacturing Services.

ManufacturingServices is a class also implementing the interface Callable. The system uses "singleton" design pattern to create and maintain a single instance of ManufacturingServices to manage all the agents in the system.

ManufacturingServices, BehaviourAgents and all the other possible class types, including IInteger, IBoolean, IFloat, IDouble are wrapped as a class. Objects of these classes will be initialized to be used in the nodes of the tree, these nodes carry the class type, in this sense, ALAN use Callable as another layer under Interpretable to provide class type information to implement the object genetic programming.

6.4 Population Initialization

As stated above, a Genetic Programming Interpreter Language is ready, with the same language features and statements as the conventional language being programmed in. And moreover, the language is based on interpreter building blocks.

These building blocks are constructed together to generate ALAN GP programs, in the structure of S-Expression binary trees whose nodes are made of two different types : function sets and terminal sets (Leaf nodes and Non-Leaf nodes).

ALAN has quite different concepts in Leaf and Non-Leaf nodes, due to its extra layer on class type definition.

- Non-Leaf nodes (referred as function sets above) of trees are objects of Sequence, IfThenElse, Loop, VarDeclaration, Assignment, inherited from abstract class Term.

- Leaf nodes (referred as terminal sets above) of s-expression trees are objects of DotOp, Literal, LocalVar, Return, inherited from abstract class Expression. Literal takes any Callabe objects as value; LocalVar takes one local variable declared already by VarDeclartion; Return takes Literals or LocalVars according to the data requirement; and DotOp takes FunctionCall as its sub-node, to get the value of function operation.

The leaf nodes in ALAN are not the real sense of tree nodes in the tree structure, since they are requiring sub-node(s), such as values of different class types. However, they are leaf nodes of the tree, functionally.

Each tree starts with *Sequence* and ends with *Return*. By calling the function "*initTree*" of *Sequence*, a tree structure was created automatically, since all the "*initTree*" functions in the randomly selected nodes are iteratively called. Each Interpretable inherited class has an "*initTree*" function:

public void initTree(int deepMax, FunctionFrame frame, Class returnType, Select s);[2]

The random selection of nodes is not absolute. During population initialization, care must be taken to create a legal program. The nodes open to be selected have to be constrained only within the pre-specified class type. Once the class type of the child node is determined, all objects of that type have an equal chance of being selected.

Each Interpretable inherited class contains the function of

public Class getReturnType();

[2] FunctionFrame is the frame we use to hold the tree structure. Each FunctionFrame is an associated program, evolved using genetic programming.

to provide the information of type of the node.

Genetically generated trees will vary in size and shape. A restriction is made on the tree's maximum depth as well. Each Interpretable node provides the function:

calculateTreeDeepness (FunctionFrame ff, int currentDeep)

to trace the depth of each node. While this depth is not reached (depth > 2), nodes are randomly chosen from the reunion of function and terminal sets. When the maximum depth is reached (depth=2), nodes are randomly chosen from the terminal set alone, which causes the tree's depth to be no greater than the maximum depth allowed.

The combinations of different nodes, i.e., interpreter building blocks compose our GP individuals. Certain numbers of individuals (depending on system GP parameters) compose our initial population (Generation 0) and get ready for GP run.

6.5 GP Operators

6.5.1 Mutation

To be truly random, the mutation node must have an equal chance of being any node in the parent tree. Both leaf node and non-leaf node have the same probability to be selected

in ALAN system. This would be done by finding the number of nodes in the tree N, picking a random number R from 1 to N, and then doing an preorder traversal of the tree through R-1 nodes.

After a mutation point is randomly chosen, the subtree rooted at that point is deleted, and a new subtree is grown there using the same random growth process that was used to generate the initial population. The following function is shared to conduct both initial population generation and mutation.

public void initTree(int deepMax, FunctionFrame frame, Class returnType, Select s);

Once again, we need to take care of the class type, in order to get the syntactically legal programs and of the depth of the tree, as well, since we don't want the memory runs out while executing the tree of too big size.

6.5.2 Crossover

In Object-Oriented GP, crossover has to be conducted between two nodes of the same class type. The crossover node (XN_{Tree1}) in one tree is first selected randomly as described in mutation (see section 6.5.1); the crossover node in the other tree (XN_{Tree2}) will be randomly selected from those nodes with the same class type as XN_{Tree1}.

The sub-trees under XN_{Tree1} and XN_{Tree2} then are swapped

public void swap(NodeDescriptor childNode1, NodeDescriptor childNode2);

to get two new programs.

In most GP applications, a certain mechanism has to be set up to make the leaf nodes and non-leaf nodes not interchangeable. As in the mathematical example, the leaf nodes were operands and the non-leaf nodes were operators. This problem is not applicable over here in ALAN, due to the class type constraints.

6.6 GP Controls

Once again, ALAN is a framework[3] to make up a reusable GP design. Inheritance and polymorphism allow easy modification of default behaviors. Changing parameters and components is typically just a matter of overriding inherited methods and altering their default behavior.

[3] A Framework is a set of cooperating classes that make up a reusable design for a specific class of software. (Gamma, et al., 1995)

ALAN, itself, is a scalable system, providing various GP control policies. By simply instantializing these pre-specified policy, depending on the users' different requirements, GP control strategies and parameters will be applied to GP run.

interface PopInitPolicy

is the mechanism to start up the initialization population (Generation 0) of GP run.

RandomInitPolicy implements PopInitPolicy. RandomInitPolicy starts up the initialization population, by changing the prototypes of an individual by mutation and inserting it into the initialization. In this way the entire population is created.

Interface GenPolicy

is responsible for creation of the next Generation.

CrowdingGenPolicy implements GenPolicy. CrowdingGenPolicy is to be used to insert the new individuals to the current population; when the population number count reaches a certain pre-set value, population is full. Genetic operators start to work on the population and the new individuals are moving to next generation.

Interface ElitePolicy

is responsible for reproduction – one of the genetic operators.

StandardElitePolicy implements ElitePolicy. StandardElitePolicy takes the number at the users' own choice (n) to pass the best n individuals as elites to the next generation.

Interface SelectionPolicy

is responsible for selecting individuals from the population and move them to a mating pool.

TournamentSelectionPolicy randomly selects individuals of size n=2 to compare their fitness, the individual with the better fitness value will be selected to mating pool.
RouletteWheelSelectionPolicy makes it happen that the individual with the better fitness will get better chance to be selected.
EliteSelectionPolicy passes a copy of the individual with the best fitness value in the current population to the new generation.

interface KillPolicy

makes sure that the population size does not exceed the maximum acceptable number in each generation. By setting another parameter as the worst fitness, the fitness could be improved by eliminating the individual with the worst fitness.

PairFightingKillPolicy is developed, to implement the interface KillPolicy. Two individuals are randomly selected and their fitness values are calculated and compared.

The individual with the worse fitness value is removed from the population. The system reduces the number of individuals in the population in this way, in "steady-state" GP.

ReverseFitnessKillPolicy: In our system, we also keep the worst fitness value in each generation. The individual with the worst fitness value will be removed from the current generation when this policy was followed.

interface TerminationPolicy

determines the abort condition for GP run.

GenNumTerminationPolicy: GP run is terminated when the self-set generation number has been reached.

NoChangeTerminationPolicy: An array is used to hold the fitness history. When the best fitness remains the same over a certain number of generations (convergence might happen), this policy will terminate the GP run.

FitnessFulfillTerminationPolicy: When the best fitness meets the problem requirement, GP run stops.

interface ExplorationPolicy

NoChangeExplorationPolicy implements ExplorationPolicy. The function of *adaptVariationParameters()* adapts the mutation and crossover rate in GP run, when the best fitness over a certain number of generations keep the same.

It is a creative idea that during a GP run, genetic operator parameters is to be modified dynamically. The strategy for the modification of the parameters is described by the interface ExplorationPolicy with the implemented class NoChangeExplorationPolicy. The policy will check whether the best fitnesses in the last ten generations is improved. If the fitness remains the same for long, the current crossover and mutation rate is increased by a certain amount (10%, for example). This process continues, until the fitness of the best individual get improved. The crossover and mutation rate will be reset again to its original value, after this process is done.

6.7 ALAN Fitness Function

Fitness function is a very significant element in Genetic Programming. The fitness function rates the performance of a possible solution.

The purpose of ALAN is to obtain a feasible plan in the Manufacturing Services system, how and when to process each operation of each order on different resource machines to make sure that each order will be complete on the due date.

For the purposes of this system, a fitness function based on minimizing the sum of the square of the error (SSE) is used in ALAN:

$$f = \sum_{i=1}^{n} (T_{CompleteTime} - T_{DueDate})^2$$

The deviation of each order's complete time and its due date is enlarged by square of the error. Simulation:

- $T_{CompleteTime}$ is the time one order needs to be released from machine. It is obtained by simulation. In ALAN, simulation work with fitness function to assign fitness value to each individual.
- $T_{DueDate}$ is the time required by the customers for each order.
- n is the number of orders in the system.

The lower fitness one candidate solution gets, the better performance the solution presents.

In Genetic Programming, program is evaluated for fitness by executing the program. The function of *interpret()* in each *Interpretable* class will recursively traverse the tree top-down and execute the program, acting on each statement as it reaches.

The fitness evaluation in ALAN is carried out parallel. Systems take those individuals coming back from EvalServer, with the fitness value assigned to conduct genetic operations. For those individuals which take too long, there might be infinite execution loops. System sets certain time to wait, after that time, system is going to „kill" this individual to prevent program growing out of stack.

Chapter 7

Results, Discussion Issues and Future work

7.1 Results

7.1.1 Test Problem Results

Target of the first phase of ALAN was to build the prototype and examine the possibilities for the creation and optimization of behavior strategies by means of genetic programming on the conceived infrastructure.

The test problem was developed – Traveling Salesman Problem (TSP) was used to examine the GA operation and a two-dimensional function - " blind mountain climber " was set to examine GP operations in the system, defined as the following function:

$$y = e^{-|x-a|} * \sin(5*(x+a)) \qquad \textit{(Equation 7-1)}$$

with: $y = f_a(x)$ *function value at the point* x

x *Position*

a *band factor (test environment)*

The variable a is ranged within

$$-3 \le a \le 3 \qquad \textbf{(Equation 7-2)}$$

The experiment use GP to coordinate between x and a to reach the target - get the max value of y. Figure 6-1 shows two functions with $a = 0$ and $a = 3$.

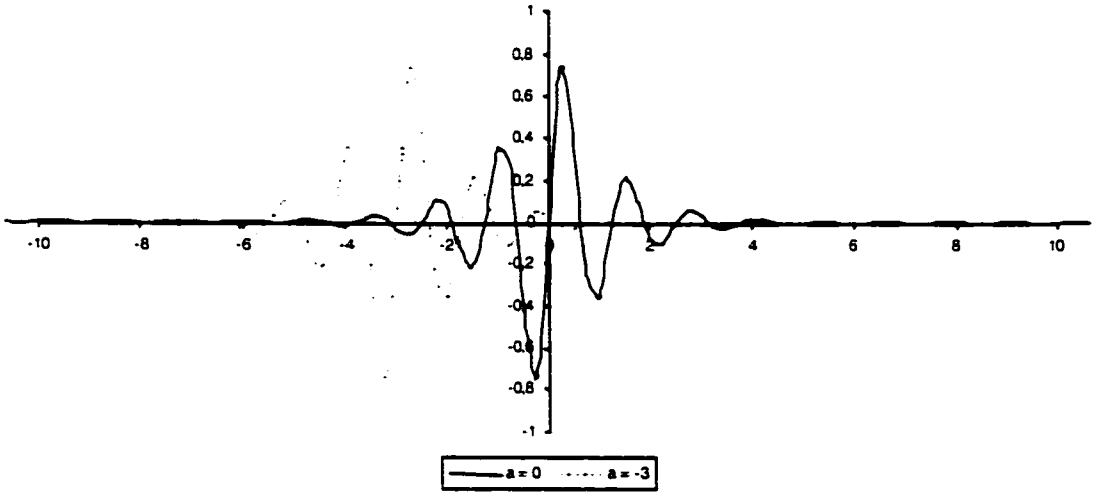

Figure 7-1: Function Band $f_a(x)$ with a=0 and a=-3

The fitness is calculated like in equation 1-3:

$$fitness = 1 - f_a(x) \qquad \textbf{(equation 7-3)}$$

Also for each test the initial population was set to 40 individuals – 10 individuals per protoype. The size of the *MatingPool* was set to 50% of the population, which means 20 individuals are allowed to be selected into *MatingPool* to conduct genetic operations – 10% of them are reproduced, 10% of them are mutated and 30% are recombined. Meanwhile, 20 individuals from the previous generation are killed to maintain the "steady-state" of the population. Thus the next generation again consists of 40 individuals and maintains constant population size.

As shown in figure 6.2, the fitness of the static function (when a=0) is approaching to its best value after 40 generations (*fitness* = $1 - f_0(x) = 1 - 0.745 = 0.255$). In this case, no variable strategy is developed.

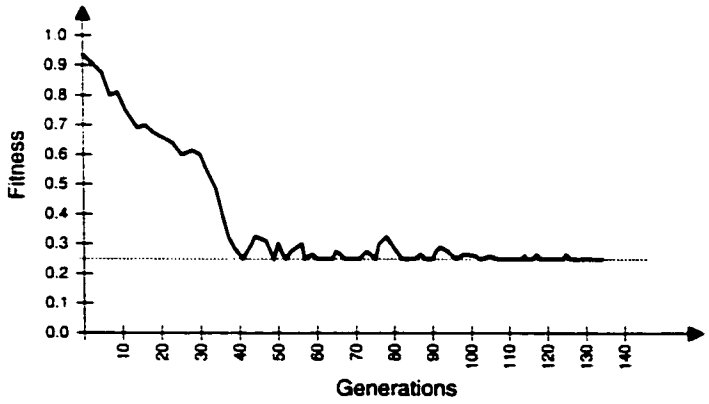

Figure 7-2: Fitness of a Static Function with a=0

Figure 7-2 shows the best fitness of each generation for the function with *a* as its variable. The best fitness (*fitness* = $1 - f_0(x) = 1 - 0.590 = 0.410 > 0.255$) was not reached by the developed individuals.

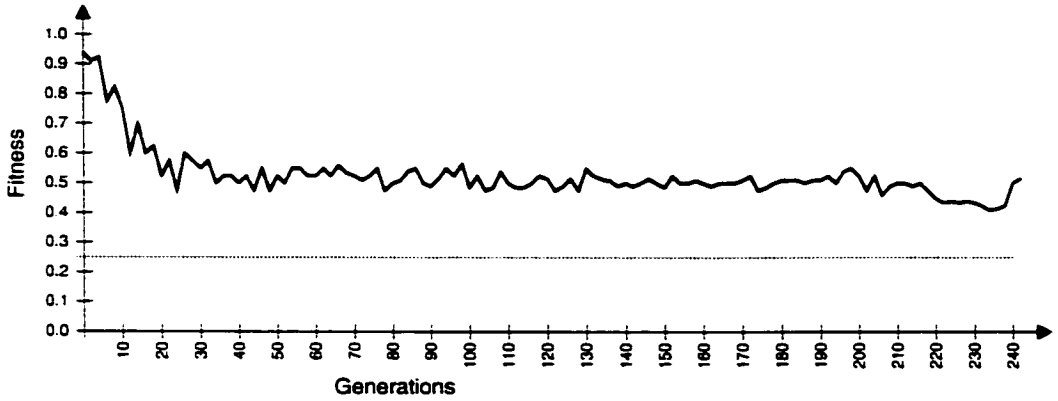

Figure 7-3: Fitness of Individuals Generating Variable Strategies

The test problem shows that GP is able to improve preset basic prototype strategies and thus optimize the behaviour of the individuals. However, GP is not able to develop completely new strategies, nevertheless the adaptation and improvement of the basic prototype strategy was beyond of pure parameter variation.

7.1.2 ALAN Experiment Results

Tournament selection, defined in TournamentSelectionPolicy, was used in ALAN experiments. The individual with the better fitness (smaller value) is considered to be the winner and is copied into a mating pool with the same size as the initial population. After the mating pool is full, reproduction is applied to 10% of the individuals, mutation to 5% of the individuals and crossover to the remaining ones. The individuals resulting from the operator application are copied into a new population. The process ends either at a limit generation (according to GenNumTerminationPolicy) or when an individual with some

goal fitness found (following FitnessFulfilTerminationPolicy), or convergence occurs (based on NoChangeTerminationPolicy).

As I was assigned a one year contract with IPA, at the time of my departure, the ALAN team was still working on the first phase – prototype. The whole integrated system results are yet to be finalized. But it has already been clearly proved by the previous test problems that genetic programming is able to optimize behavior strategies and therefore the learning strategy for the order and resource agents is about to be discovered.

With the turn of the year, ALAN is starting its second implementation phase on order and resource agent, an abstract semi-conductor manufacturing environment, a virtual market between the agents. A framework is about to be implemented for the negotiation strategies between the agents and the target systems of the agents.

During my stay at IPA, I was mainly responsible for developing the Interpreter package for the ALAN system. Here, I would like to give some crossover and mutation programs as a result, (as the AgentSystem package was to be completed in the second phase, the execution of tree was difficult) and discuss some related issues arising during the development and testing of Interpreter.

Parent Tree 1:

```
IDouble lillyTest(IInterger a, IDouble b){

        IBoolean unique_1 = true;
        OrderAgent unique_12 =
                de.fhg.ipa.alan.agentSystem.OrderAgent@3a9f2fe7;
        ResourceAgent unique_11 =
                de.fhg.ipa.alan.agentSystem.ResourceAgent@3b7f2fe7;
        OrderAgent unique_13 = de.fhg.ipa.alan.agentSystem.OrderAgent@3a1f2fe7;
        IInterger unique_9 = -4;
        IDouble unique_3 = 1.9509055559440363;
        ManufacturingServices ms =
                de.fhg.ipa.alan.agentSystem.ManufacturingServices@3b8e9d93;
        ....
        if (((-0.326596197897997).greater(b)).not()){
                unique_11 = unique_12.getCheapestResourceAvailable();
                [redacted]
                        [redacted]
                [redacted]
                        [redacted]
                        [redacted]
                [redacted]
        }
        else{
                unique_12.tryMoveTo(unique_11);
        }
        ....
        Return unique_3;
}
```

Parent Tree 2:

```
IDouble lillyTest(IInterger a, IDouble b){

        IRandomizer unique_1 =
                de.fhg.ipa.alan.agentSystem.Irandomizer@935f2fef;
        IFloat unique_7 = 0.0;
        IInterger unique_13 = 9;
        OrderAgent unique_5 =
                de.fhg.ipa.alan.agentSystem.OrderAgent@370e2868;
        ResourceAgent unique_10 =
                de.fhg.ipa.alan.agentSystem.ResourceAgent@40ae286a;
        ResourceAgent unique_12 =
                de.fhg.ipa.alan.agentSystem.ResourceAgent@40ae286a;
        ManufacturingServices ms =
                de.fhg.ipa.alan.agentSystem.ManufacturingServices@3f8e2868;
        IInteger  unique_8 = -7;
        IBoolean unique_3 = true;
        IDouble unique_14 = 2.780778232865;
        ....
        for (unique_9 = de.fhg.ipa.alan.agentSystem.Irandomizer@b716286a.
                getIInteger(); unique_9.lessthan(unique_13); unique_9++){
                [redacted]
                        [redacted]
                unique_3 = true;
                if (unique_10.sell(unique_12)){
                        unique_13.equalTo(unique_9));
                else{
                        ms.getMS();
                }
        }
        ....
        Return –3.818988636005816;
}
```

Crossovered Tree 1:

```
IDouble lillyTest(IInterger a, IDouble b){

        IBoolean unique_1 = true;
        OrderAgent unique_12 =
                de.fhg.ipa.alan.agentSystem.OrderAgent@3a9f2fe7;
        ResourceAgent unique_11 =
                de.fhg.ipa.alan.agentSystem.ResourceAgent@3b7f2fe7;
        OrderAgent unique_13 = de.fhg.ipa.alan.agentSystem.OrderAgent@3a1f2fe7;
        IInterger unique_9 = -4;
        IDouble unique_3 = 1.9509055559440363;
        ManufacturingServices ms =
                de.fhg.ipa.alan.agentSystem.ManufacturingServics@3b8e9d93;
        ....
        if (((-0.326596197897997).greater(b)).not()){
                unique_11 = unique_12.getCheapestResourceAvailable();
                [redacted]
        }
        else{
                unique_12.tryMoveTo(unique_11);
        }
        ....
        Return unique_3;
}
```

The Shaded parts are crossovered. Since the crossover point in the Parent Tree 1 is a *Term*. The *Terms* of any kind in Parent Tree 2 get equal chances to be selected. An *Assignment* is selected in this experimental case.

Crossovered Tree 2:

```
IDouble lillyTest(IInterger a, IDouble b){

        IRandomizer unique_1 =
                de.fhg.ipa.alan.agentSystem.Irandomizer@935f2fef;
        IFloat unique_7 = 0.0;
        IInterger unique_13 = 9;
        OrderAgent unique_5 =
                de.fhg.ipa.alan.agentSystem.OrderAgent@370e2868;
        ResourceAgent unique_10 =
                de.fhg.ipa.alan.agentSystem.ResourceAgent@40ae286a;
        ResourceAgent unique_12 =
                de.fhg.ipa.alan.agentSystem.ResourceAgent@40ae286a;
        ManufacturingServices ms =
                de.fhg.ipa.alan.agentSystem.ManufacturingServices@3f8e2868;
        IInteger  unique_8 = -7;
        IBoolean unique_3 = true;
        IDouble unique_14 = 2.780778232865;
        ....
        for (unique_9 = de.fhg.ipa.alan.agentSystem.Irandomizer@b716286a.
                getIInteger(); unique_9.lessthan(unique_13); unique_9++){
                [redacted]
                unique_3 = true;
                if (unique_10.sell(unique_12)){
                        unique_13.equalTo(unique_9));
                else{
                        ms.getMS();
                }
        }
        ....
        Return -3.818988636005816;
}
```

a Tree Before Mutation:

```
ResourceAgent aOrderAgentOperation(IInterger int){
        ManufacturingServics ms =
                de.fhg.ipa.alan.agentSystem.ManufacturingServices@f91caeff;
        OrderAgent unique_5 =
                de.fhg.ipa.alan.agentSystem.OrderAgent@bc5cae0a;
        IInterger unique_6 = 5;
        ResourceAgent unique_4 =
                de.fhg.ipa.alan.agentSystem.ResourceAgent@cb7caef7;
        IBoolean unique_7 = false;
        IDouble unique_9 = 6.36822151476540;
        IBoolean unique_8 = false;
        ....
        this.standardActivateInBuffer();
        if (unique_7.equalTo(unique_8)){
                int ms.getTimeCurrentOrderReady(unique_4);
        }
        else{
                return this.getCurrentResourceAgent();
        }
        return unique_4;
}
```

The Tree After Mutation:

```
ResourceAgent aOrderAgentOperation(IInterger int){
        ManufacturingServics ms =
                de.fhg.ipa.alan.agentSystem.ManufacturingServices@f91caeff;
        OrderAgent unique_5 =
                de.fhg.ipa.alan.agentSystem.OrderAgent@bc5cae0a;
        IInterger unique_6 = 5;
        ResourceAgent unique_4 =
                de.fhg.ipa.alan.agentSystem.ResourceAgent@cb7caef7;
        IBoolean unique_7 = false;
        IDouble unique_9 = 6.36822151476540;
        IBoolean unique_8 = false;
        ....
        this.standardActivateInBuffer();
        if (unique_7.equalTo(unique_8)){
                [redacted]
                        [redacted]
        }
        else{
                return this.getCurrentResourceAgent();
        }
        return unique_4;
}
```

The mutation point is randomly selected and happens to be the *Assignment*. The parent tree has an assignment to *IInterger* class type; during mutation, ResourceAgent class type is selected to do Assignment.

7.2 Discussion Issues

7.2.1 GA/GP Migration

Object-oriented GP design greatly increases the complexity of GP. In addition, the complex interactions between agents, parallel simulation to calculate fitness, Database consistency layer, make ALAN a very complex system. How can ALAN people handle the complexity of this system, has become the main concern for next phase of ALAN development.

ALAN architect has to consider reducing the system complexity. In addition, a large number of GP tree program execution makes big memory consumption. GA/GP migration (Howard & D'Angelo, 1995) has been porposed and already successfully implemented in the current system.

For those numeric parameters of Manufacturing Services, ALAN uses Genetic Algorithm genomes to represent them. Agent behavior, focus of this research, represented in GP programs, becomes one gene of the GA genomes. The gene of GP takes part in the GA genetic operations, which means, this gene might be selected for mutation and crossover with the same gene in another individual; meanwhile, it has its own genetic operation, which follows the same process as described as above.

The numeric parameters in the system, such as:

totalAmountOfMoney

waitingTimeWeight

priceWeight

deltaPriceInc

deltaPriceDec

maxPrice

minPrice

will be operated based on GA and no class type is needed, so that problem get simplified at a large extent.

7.2.2 Random Control

ALAN GP initialization and various selection strategy, including selection of crossover points, mutation points, selection of individuals, Interpretable nodes, are following the complete random principle, trying to be truly random, which ends up completely different situation in each GP run. For the first phase of ALAN prototypes, system uses random seeds to generate random number, to make program debugging and analysis possible.

From the print out of the genetically created programs, we have found some illogical, naturally-generated codes, for example,

```
if ((aDoubleParamA = new Idouble(8.519369)).greater(aDoubleParamA)){
        ((((ms).getMS()).getMS()).getMS()).getOrderAgent();
        aVariable = (aResourceAgent).getPriceForOffer(-4);
}
```

Some GP automatically generated codes do not make too much sense and waste the problem search space. Therefore, another set of selection principle "SemanticalSeletion" is added into the system. Effort was made to make codes more semantically meaningful, with syntactically correct as a necessity.

7.2.3 ADFs

FunctionFrame is the framework we apply GP to discover the agent behaviour. FunctionFrame has to have the function signature, as a regular function in conventional programming language, including associated parameters (both types and names) and return type.

It is working like an operation of BehaviourAgent object. ALAN system allows an automatically generated FunctionFrame (program) to call anther automatically generated

FunctionFrame, even between simultaneously generated FunctionFrames. This is done by registering each FunctionFrame in the CallStubMap, which is a HashTable to contain all the operations of each class. Each operation has a callStub in GP Interpreter language, including Java languages functions and human specified functions. All kinds of functions – Java functions, pre-specified system functions, automatically generated GP functions have the same chance to be selected in GP run.

This is a different way of implementation of Koza (1999)'s Automatically defined Function. It is a kind of generalization and expandation.

7.3 Future Work

The work on ALAN system and other PPS approaches, such as MRP I/II, OPT, Kanban, 3-Litter PPS, will be followed and extensively researched at Fraunhofer – IPA institute years to come. A couple of PhD dissertations are in the chanel on the above topic.

Concerning with the GP part, in ALAN complex problem domain, genotype-phenotype mapping is not possibly in one-to-one model any more. The observation, comparison and improvement of the final system behaviour may be of difficulty, due to the large amount of random mechanism set inside the system. Reducing system complexity and simplifying the initial prototype, production scenario and environment is the work about to do in the near future.

In GP run, measure needs to be taken to encourage population diversity. Tournament selection, reproduction and crossover appear to converge too readily. Mutation is largely inapplicable to genetic programming (Koza 1992)[4]. Simply choosing an easier evaluation function might not be the best solution in complex problems. Dynamic evaluation function scaled over different time (Fukunaga & Kahng, 1996) might be an approach to better performance and need to be discovered in the future.

[4] In genetic programming, particular functions and terminals are not associated with fixed positions in a fixed structure.

Chapter 8
Conclusion

Besides PPC system, numerous enterprises use new tools for modeling, simulation, and optimization in production planning. These tools facilitate good modeling of business operations, as well as using the latest methods in the search for solutions. ALAN is making effort to offer a new method to implement production planning automation and adaptation in a higher level – Method Selection Level.

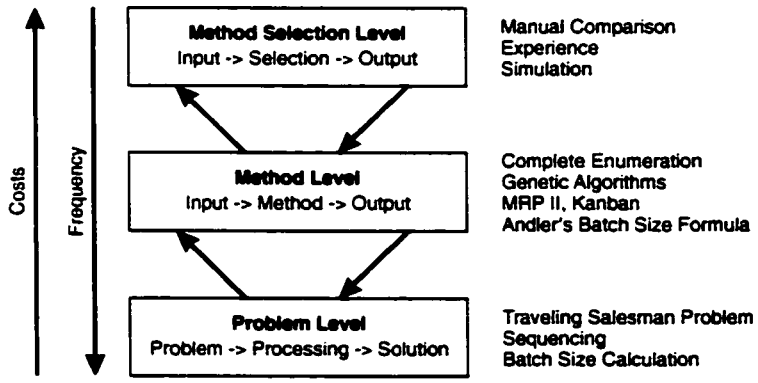

Figure 8-1 Levels of Optimization

An altogether new approach is the optimization of the planning methods used by means of known optimization methods on the method selection level. This requires a new combination of methods, and promises significant improvements of the ability to adequately model enterprises, the PP&C methods themselves and the user interface of future systems. The approach presented in ALAN combines known planning and

optimization methods into an automatically optimized adaptive controller consisting of modelable controller elements. The adaptability inherent in the methods simultaneously removes current systems' shortcomings: the lack of flexibility, adaptability and cooperative planning.

Genetic Programming has recently emerged as an important paradigm for automatic generation of computer programs. GP has been extensively applied to generate functional and procedural programs in various application areas. Since object-orientation is becoming one of the more heavily used methodologies for software engineering, some experimentation is done in this research to apply Genetic Programming into automatically generation of object-oriented programs. It is of critical importance to the acceptance of genetic programming to be extended to work in this area.

References

[1] Aler, R., Borrajo, D., Isasi, P. (1998), *Genetic Programming of Control Knowledge for Planning,* American Association for Artificial Intelligence

[2] Banzhaf, W., Nordin, P., Keller, R.E. & Francone, F.D.(1998), *Genetic Programming – An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers, Inc., San Franciscon, California.

[3] Becker, D. (1991), *Gegenstandsorientiertes Simulationssystem mit parametrisierter Netzwerkmodellierung für Fertigungsprozesse mit Stückgutcharakter*, IPA – IAO Forschung und Praxis 154 Berlin, Heidelberg: Springer, 1991. Zugl. Diss., Universität Stuttgart, 1991

[4] Bergé, P. (1984), *Order within Chaos, Towards a deterministic approach within chaos*, Paris: John Wiley & Sons, 1984

[5] Bongaerts, L.; Wyns, J.; Detand, J.; Van Brussel, H. (1996), *Identification of Manufacturing Holons*, In: Proceedings of the European Workshop on Agent-Oriented Systems in Manufacturing, Berlin: 26-27,9,1996

[6] Bruce, W.S.(1995), *The Application of Genetic Programming to the Automatic Generation of Object-Oriented Programs*, Nova Southeastern University 1995

[7] Bruns, R. (1996), *Wissensbasierte Genetische Algorithmen: Integration von Genetischen Algorithmen und Constraint-Programmierung zur Lösung kompinatorischer Optimierungsprobleme*, Zugl. Oldenburg, Universität, Diss., 1996.

[8] Chong, Fuey Sian and Langdon, W. B. (1999), *Java based distributed genetic programming on the internet*, In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith (editors), *Proceedings of the Genetic and Evolutionary Computation Conference* , volume 2, page 1229, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[9] Cramer & Nichael L. (1985). *A representation for the adaptive generation of simple sequential programs.* In John J. Grafenstette (editor), Proceedings of an International Conference on Genetic Algorithms and the Applications , pp. 183-187, 1985.

[10] Davis, L. (1985), *Job Shop Scheduling with Genetic Algorithms*, Proceedings of the International Conference on Genetic Algorithms and Their Applications 136-140, Pittsburgh, PA

[11] Fukunaga L. A. S. & Kahng, A.B. (1996), *Improving the Performance of Evolutionary Optimization by Dynamically Scaling the Evaluation Function*

[12] Gen, M. & Cheng, R. (1997), *Genetic Algorithms and Engineering Design*, John Wiley & Sons, NY, USA.

[13] Goldberg, David E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

[14] Holland, J.H. (1975). *Adaption in natural and artificial systems*. Ann Arbor: University of Michigan Press.

[15] Howard L. M. & D'Angelo, D. J. (1995) "*the GA-P: A Genetic Algorithm and Genetic Programming Hybrid*". IEEE Expert. Vol 10, N. 3, June 1995.

[16] Ito, T., Iba, H. & Sato, S. (1999), *A Self-Tuning Mechanism for Depth-Dependent Crossover, Advances in Programming 3*, MIT Press, pp.377-399, 1999

[17] Iwata, K. & Onosato, M. (1994), *Random Manufacturing System: a New Concept of Manufactuing Systems for Production to Order*, Annals of the CIRP 43 (1) 1994, S 379-384

[18] Kimura, F. (1993), *A Product and Process Model for Vitual Manufacturing Systems*, Annals of the CIRP 42 (1), 1993, 147-150

[19] Kinnear, Kenneth E., Jr. ,editor (1994). *Advances in Genetic*. Cambridge, MA: MIT Press.

[20] Kobayashi, S., Ono, I. & Yamamura, M. (1995), *An Efficient Genetic Algorithm for Job Shop Scheduling Problems*, Proceedings of 6th International Conference of Genetic Algorithm, 506-511

[21] Koza, J.R. (1992), *Genetic Programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press.

[22] Koza, J.R. (1994), *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press.

[23] Koza, J.R. (1995), *Gene duplication to enable genetic programming to concurrently evolve both the architecture and the work performing steps of a computer program*, In Proceedings of the 1995 International Joint Conference in Artificial Intelligence.

[24] Koza, J.R. and Andre, D. (1996), *Evolution of iteration in genetic programming*, In Evolutionary Programming V: Proceedings of the Fifth Annual Conference on Evolutionary Programming. Cambridge, MA: MIT Press.

[25] Koza, J.R., Bennett III, Forrest H., Andre, D. & Keane, M.A. (1999), *Genetic Programming III – Darwinian Invention and Problem Solving.* Morgan Kaufmann Publishers, Inc., San Franciscon, California.

[26] Luke, S. and Spector, L. (1997), *A comparison of Crossover and Mutation in Genetic Programming*, Proceedings of Genetic Programming 1997, pp.240-248, 1997.

[27] Montana, D.J. (1994), *Strongly Typed Genetic Programming*, Technical Report 7866. Bolt Beranek and Newman, Inc., March 25, 1994

[28] Sarafopoulos, A. (1999), *Automatic Generation of Affine IFS and Strongly Typed Genetic Programming* in Second European Workshop, EuroGP'99, Göteborg, Sweden, May 1999 Proceedings

[29] Shi, G., Iima, H. & Sannomiya, N. (1997), *Comparison of Two Genetic Algorithms in Sovling Tough Job Shop Scheduling Problems*, Trans. IEEE of Japan 117-C (7), 856-864

[30] Taillard, E.D. (1994), *Parallel Taboo Search Techniques for the Job Shop Scheduling Problem*, ORSA Journal on Computing 6 (2): 108-117

[31] Ueda, K. (1993). *A genetic approach toward future manufacturing systems*; in Peklenik, J. (editor), Flexible Manufacturing Systems, Past, Present and Future, CIRP, faculty of Mechanical Engineering, Ljubljana pp. 211-228

[32] Vaessens, R.J.M., Aarts, E.H.L. (1996), *Job Shop Scheduling by Local Search*, INFORMS Journal on Computing, 8, 1996, pp.302-317

[33] Warnecke,H.-J.(1993), *CIM out – Lean in?* In: Univ. des Saarlandes, Institut fuer Wirtschaftsinformatik CIM im Mittelstand. Fachtagung. Schlanke Produktion im Schlanken Unternehmen, Saarbruecken: 1993, S.1-16

[34] Westkämper, E., Balve, P. & Wiendahl , H. (1998), *Auftragsmanagement in wandlungsfähigen Unternehmensstrukturen - Anforderungen und Ansätze*, In: PPS Management 3 (1998) 1, S.22-26

[35] Winter, C.S., McIlroy, P.W.A. & Fernandez-Villacana, J.L. (1994), *Evolving Software Techniques*, BT Technology Journal 12(2) 121-131

[36] Yamada, T. & Nakano, R. (1995), *A Genetic Algorithm with Multi-Step Crossover for Job-Shop Scheduling Problems,* First IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA ,95) 146-151, Shieffield, UK

[37] Yamada, T. & Nakano, R. (1996), *A Fusion of Crossover and Local Search,* IEEE International Conference on Industrial Technology (ICIT, ,96), 426-430, Shanghai, China

[38] Zalzala, A.M.S. and Green, D. (1999), *MTGP: A multithreaded Java tool for genetic programming applications,* In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao, and Ali Zalzala (editors), Proceedings of the Congress on Evolutionary Computation , Volume 2, pp. 904-912, Mayflower Hotel, Washington D.C., USA, 6-9 July 1999. IEEE Press.

Appendix: A GA/GP Glossary

Building Block: A pattern of genes in a contiguous section of a chromosome.

Chromosome: The bit string (GA) or program parse tree (GP), which represents the individual.

Closure: all the variables, constants, arguments for functions, and values returned from functions must be of the same data type.

Convergence: Tendency of members of a population to be the same.

Crossover: The genetic process by which genetic material is exchanged between individuals in the population.

Elitist: GA which always retains the best individual in the population found so far (Tournament selection is naturally elitist).

Evolution Strategy: A search technique, where the next point to search is given by adding gaussian random noise to the current search point.

Fitness Function: Function which evaluates a member of a population.

Function Set: The set of operators used in GP, these functions label the internal (non-leaf) points of the parse trees that represent the programs in the population. An example function set might be {+, -, *}.

Generation: An iteration of the measurement of fitness and the creation of a new population by means of genetic operations.

Genetic Algorithm (GA): Model of machine learning that uses a genetic/evolutionary metaphor. Implementations typically use fixed-length character strings to represent their genetic information.

Genetic Operator: An operator in a genetic algorithm or genetic programming, which acts upon the chromosome to produce a new individual. Example operators are mutation and crossover.

Genetic Programming (GP): Genetic Algorithms applied to programs. Genetic Programming is more expressive than fixed-length character string GA's, though GA's are likely to be more efficient for some classes of problems.

Genotype: "Physiological Team" in which a gene can make a maximum contribution to fitness by elaborating its chemical "gene product" in the needed quantity and at the appropriate stage of development.

Mutation: Arbitrary change to representation, often at random. In Genetic Programming, a subtree is replaced by another, some or all of which is created at random.

Non-Terminal Functions: This name may be used to avoid confusion with functions with no parameters which can only act as end points of the parse tree (i.e. leafs) and so are part of the terminal set.

Phenotype: Product of the interaction of all genes.

Premature Convergence: When a Genetic Algorithm's population converges to something which is not the solution you wanted.

Reproduction: The genetic operation which causes an exact copy of the genetic representation of an individual to be made in the population.

Terminal Set: The set of terminal (leaf) nodes in the parse trees representing the programs in the population. A terminal might be a variable, such as X, a constant value, such as 42, or a function taking no arguments, such as (move-north).

Tournament Selection: A mechanism for choosing individuals from a population. A group are selected at random from the population and the best (normally only one, but possibly more) is chosen.

Appendix B: Fraunhofer IPA Introduction

The Research Organization

The Fraunhofer-Gesellschaft is the leading organization of applied research in Germany. It currently operates 48 research institutes at 38 locations throughout Germany, with about 9,600 employees, most of them are scientists and engineers. The annual research budget amounts to over 1.5 billion German marks since 1999.

International activities are increasingly important. Apart from the collaboration with numerous companies and research establishments within Europe, the Fraunhofer-Gesellschaft operates resource centers and research units in Asia and the United States. Revenue generated by the Fraunhofer-Gesellschaft abroad exceeded 107 million DM in the year of 1998.

The Fraunhofer-Gesellschaft was founded in 1949 and is a recognized non-profit organization. Amongst its members are well-known companies and private patrons who contribute to the promotion of its application-oriented policy. The name Fraunhofer-Gesellschaft was chosen in reference to the successful Munich researcher, inventor and entrepreneur Joseph von Fraunhofer (1787-1826), who won high acclaim for his scientific and commercial achievements.

The Research Fields

Research conducted by the Fraunhofer-Gesellschaft focuses on specific tasks across a wide spectrum of research fields, where systematic solutions are required, several institutes collaborate on an interdisciplinary basis, such as:

- Materials technology, components
- Production technology
- Information and communications technology
- Microelectronics, microsystems technology
- Sensor systems, testing and measurement technology
- Process engineering
- Energy and construction technology
- Environmental and health research
- Technical and economic studies, information transfer

IPA

FhG-IPA (Fraunhofer-Institute for Manufacturing Engineering and Automation) is one of the 48 Fraunhofer institutes. They are to provide a flow of know-how and technology from the researcher in the university environments to industrial enterprises.

In 1959 IPA was founded in Stuttgart and has more than 200 scientists today, who work in 7,700 m2 of offices, laboratories and outdoor facilities. The annual turnover amounted to a total of DM 66 millions in 1998. 1999 is the big jubilee year, when the Fraunhofer Society is celebrating its 50th birthday.

IPA with its 200+ scientists and a turnover in excess of 23 MECU works in various areas of control of manufacturing networks, enterprise organization, manufacturing engineering and electronic commerce.

In particular the section "Enterprise Development and Logistics" covers strategic planning projects concerning the optimization of manufacturing networks in terms of order and information flows, Internet-applications for logistics, machine learning and negotiation-based order flow optimization. From the experiences gained in these projects a number of software tools were developed at IPA. These tools include multi-agent systems, simulation (strategic and operational), capacity planning and shop floor control. By the development of these tools know-how in the area manufacturing optimization was built up at IPA and used in more than 200 industrial projects.

IPA has participated in numerous ESPRIT and other EC-funded projects before, most notably in the projects 5478 SHOP-CONTROL, 8865 Real-I-CIM and 20544 X-CITTIC. IPA therefore has lots of positive experiences in working together with other European enterprises and research facilities.

Vita Auctoris

Huining Yuan is a M.S. candidate in the School of Computer Science, University of Windsor, Ontario, Canada. She graduated with a B.S. in Computer Science from Jilin University, China in 1993. This thesis work was initialized in Stuttgart, Germany, where she was selected to participate in the University of Windsor / Fraunhofer (IPA) Internship Program. This internship program was established to provide the outstanding graduate students at the University of Windsor with an exciting opportunity to carry out the research in a major well-reputed international research institution.