# 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. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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.

**UMI**

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

# NOTE TO USERS

**The original manuscript received by UMI contains pages with indistinct print. Pages were microfilmed as received.**

**This reproduction is the best copy available**

**UMI**

Université d'Ottawa · University of Ottawa

# A Multi-Agent Extension of Negoplan and Its Application to a Business Strategy Game

by

Emre Erkol

School of Information Technology and Engineering
University of Ottawa

A thesis submitted to
the School of Graduate Studies and Research and
the Ottawa-Carleton Institute for Computer Science
in partial fulfilment of the requirements
for the degree of
Master of Computer Science

© Emre Erkol, Ottawa, Canada, 1998.

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*

*Our file Notre référence*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-32534-2

Canada

## ACKNOWLEDGEMENTS

For their supports, comments and suggestions on my thesis, I would like to thank:

especially to Stan Szpakowicz who supervised my thesis and supported me from the beginning to the end. His efforts to improve my work and his unbelievably short response time to questions asked by e-mail will never be forgotten. I heard of no supervisor who would return back the chapters to their students after reading and commenting in a couple of days, sometimes in the same day.

to Gregory Kersten who supervised the managerial aspects of my thesis. Also, his appreciative manner towards my thesis in every correspondence highly encouraged me.

to my defense committee members, Stan Matwin, Franz Oppacher.

to Terry Copeck who generously shared with me his experience on the details of the Negoplan's interface.

For their moral supports, I would like thank to my parents, Nimet and Erol Erkol, to my sister, Yelda Ören, and to my brother in law, Tuncer Ören. For their encouragement and their supports, in pursuing my education in Canada, I would like to thank to my parents in law, Ülku and Somer Ataca, and to my sister in law, Bilge Ataca.

Finally, I would like to thank mostly to my wife, Ebru Ataca Erkol, and dedicate my thesis to her for always being with me during this long and hard two and a half years and for sharing my up and down moments with a lot of patience.

## ABSTRACT

Negoplan is a domain-independent decision support system which can be customised for different application domains. Negoplan enables the representation, simulation and support of sequential, context-dependent decision-making problems that involve the interaction between two participants and their decision environment.

This thesis presents extensions that increase the expressiveness of Negoplan's constructs. They enable Negoplan to represent decision-making problems that involve more than two participants. This enlarges substantially the class of Negoplan's applications.

The usefulness of the extensions is demonstrated on a managerial decision-making application that involves more than two participants. This Negoplan application concerns self-training of a Business Administration student to test his managerial decision-making skills in an environment in which Negoplan represents two other autonomously behaving participants. The application also demonstrates the suitability of customised Negoplan as an automated training tool.

# TABLE OF CONTENTS

## TABLE OF FIGURES

# 1. INTRODUCTION

In many fields of study, the education of individuals is ideally achieved by the teaching of theory followed by a practical application of this theory. Those who have enough background knowledge to understand a new discipline are first taught the theory of that discipline. Then, they put their theoretical knowledge into practice in supervised environments and are provided feedback about the application of this knowledge.

Theoretical knowledge, in general, is taught in classrooms through presentations led by an instructor. While finding skilled instructors is usually feasible, practical education may be a challenge in some fields of study.

For example, in mathematics or finance, where the problems in general have solutions that can be proven, one can exercise theoretical knowledge by solving problems and comparing the results with the correct answers. In these types of studies, the practical knowledge of those individuals can easily be tested. However, in medical or managerial studies, where the correctness of a solution can not be proven and in which the correctness of a solution is determined by considering many different criteria, it may not be easy to find individuals and environments for practical learning. In such fields, sometimes it may not be possible to improve the practical skills before the students get into the real life problems. When decisions are unrecoverable, the lack of practical education may produce unwanted results. That is why prototypes or other artificial training tools that imitate realistic environments are being used to let individuals practice their theoretical knowledge before they get involved with real life problems.

The Negoplan system (Szpakowicz *et al.*, 1990) is a decision support system used to simulate decision-making problems. Such problems involve a decision-maker surrounded by other participants that influence his decisions. Negoplan represents this kind of problems from the viewpoint of the decision-maker and simulates the action-reaction type relations between the decision-maker and the other participants. In a simulation, if the control of the decision-making process of the decision-maker is given to a user, Negoplan becomes a training tool that lets the user observe the possible reactions resulting from different sequence of decisions.

The medical decision-making problem application (Kersten *et al.*, 1994a) has been the most important application of Negoplan system to date. This application has been used as a training tool, which addresses the lack of practical education in medical studies and provides an environment in which the medical students can apply their theoretical

knowledge in practice without requiring actual patients. This Negoplan application has successfully shown that the system can assist the practical education in medical schools.

The simulation of a decision-making problem can be implemented in two ways: the participants can be represented individually or they can be represented as part of the environment surrounding the decision-maker. In the former approach, the states of the individual participants must be distinguished and the participants' reactions to particular decisions of the decision-maker should be identified and defined separately. However, in the latter approach, the expected changes in the state of the decision-maker resulting from different decisions need to be identified and defined in the simulation, rather than representing each participant individually.

These two methods can be equally effective in the simulation of a decision-making problem, which will be used as a training tool. That is because they can both inform the user of the validity or invalidity of a series of decisions. However, this similarity may manifest itself only in the functionality of the simulations. Some simulations may be more informative if the actions taken by the individual participants and their states can also be observed from the user's point of view. This can only be achieved by representing the participants individually.

The Negoplan system provides the constructs to represent only one major participant of a decision-making problem other than the supported decision-maker. As a result of this limitation, a decision-making problem that involves more than one major participant can only be represented by treating those participants as part of the environment. One of the achievements of this thesis is an enhancement of Negoplan that allows the representation of more than one participant individually. A simulation of a managerial decision-making problem that involves more than one major participant demonstrates the usefulness of the designed and implemented extensions, though not all extensions have been actually applied.

The simulation of the managerial decision-making problem is based on the Business Strategy Game (Thompson and Stappenbeck, 1995) software (henceforth BSG) used as a practical teaching tool in the "Strategic Management and Policy II" course offered in the Master of Business Administration program at the University of Ottawa. The software provides the means to let student teams manage a manufacturing company by making annual quantitative decisions over years and compete against each other in a global market (Appendix A).

The BSG is a software that provides an environment, which is almost impossible to achieve in a teaching situation. It gives the students chance to apply their theoretical knowledge in making quantitative decisions as in real life and to get feedback quickly. In this sense, the BSG is a very useful training tool. However, its implementation that requires more than one team in order to play the game limits the use of the game only to educational institutions where naturally teams can be easily found. The purpose of simulating in Negoplan the managerial decision-making problem addressed by the BSG software is to avoid the need for other teams in an individual's practical education, thus to implement a training tool that is intelligent enough to simulate autonomously behaving teams.

Although the medical application and the BSG application of Negoplan are similar in terms of their purposes as decision-making training tools, two major differences between them make the latter worth implementing. One difference is that the former involves only one participant, while the latter involves more than one participant and makes use of the extensions of Negoplan. The other difference is that the former represents several illnesses, but none of them shows dynamic behaviour in the real sense. Therefore, every time the simulation is executed with a particular illness, the same behaviour is obtained from the system for the same sequence of decisions. One of the purposes of the BSG application is to come up with a tool that exhibits a dynamic behaviour.

Another purpose of the BSG application of Negoplan is to enable its user to make qualitative decisions, as well as quantitative decisions. The decisions of the student management teams in the actual game are only quantitative, which is unlikely to happen in real life. Real management teams usually make qualitative decisions that are translated into quantitative decisions by others. After considering this fact, the BSG application has been implemented in Negoplan in a way that enables the user to make qualitative and quantitative decisions.

Chapter 2 introduces the actual BSG software on which the managerial decision-making problem simulation is based. In this chapter we present the purpose of the software, the detailed description of all the decisions made by the student management teams and the financial indicators that show the results of these decisions.

Chapter 3 gives an overview of the Negoplan system, its constructs with their interpretation and its running mechanisms. The limitations of Negoplan's constructs that prevent the representation of more than one participant of a decision-making problem (other than the decision-maker) are also introduced in order to help understand the extensions of Negoplan.

The extensions of Negoplan that overcome the limitations of its existing constructs are presented in Chapter 4. These extensions include new constructs, modifications in Negoplan's running mechanisms and measures taken in order to prevent the inefficiency expected as a result of the increased complexity of the applications.

Chapter 5 presents the implementation of the BSG application of Negoplan using some of the extensions presented in Chapter 4. In this chapter, the representation of the participants involved in the problem and their behaviour are introduced. Section 5.2 gives the technical details of this application. Chapter 6 discusses the observations made in the implementation phase of this application regarding the suitability of Negoplan for this problem and the experimental results obtained by running this application.

Chapter 7 presents related work done on Negoplan and on systems similar to Negoplan. The Negoplan system, a domain-independent decision-support system, is considered to be an expert system shell and the related work on similar systems is based on expert system shells that have been used for training purposes.

Chapter 8 suggests some possible improvements in the extensions and in the managerial decision-making problem application. The thesis is concluded by discussing the contributions of this thesis.

The references and the bibliography of Negoplan are listed in chapters 9 and 10, respectively.

## 2. THE BUSINESS STRATEGY GAME SOFTWARE

The goal of the BSG software (Thompson and Stappenbeck, 1995) is to help prepare students for real-world decision making by giving them a chance to practice their theoretical managerial knowledge that they acquire in classroom education. The game is about managing a manufacturing company by making quantitative decisions. The realistic competitive environment with several competing companies is achieved by different teams of students taking charge of one company's business. All the companies are assumed to be in the same business, namely athletic footwear, and to be initially at the same state. The teams have full control on the management of their companies and their purpose is to bring them to the leading position in the athletic footwear industry. In order to become the industry leader, each team has to compete with each other in a growing market, and aim to get the highest global market share.

At the beginning of the game every team has two athletic footwear manufacturing plants in North America and can sell their products in all the three markets that are defined in the game: North American, European and Asian. There is a growing demand for athletic footwear products. In order to satisfy all the market demand in the following years, the teams can either expand the production capacities of their existing plants or open a new plant in Europe or Asia. The aspects of having more than one market to sell the products and being able to expand makes the game more realistic and more interesting.

The game proceeds with annual quantitative decisions of each team managing a company in the industry. The decisions are made by using the student software, (Appendix A) which is programmed to calculate the estimated results of the decisions made. After every team has made their decisions, all the decisions are collected by an evaluation software and evaluated in order to determine the actual annual results of the decisions of each team. The evaluation software is only available to the instructor and the teams do not control the evaluation process, as in real life. The teams are ranked according to their accomplishments. After having the results of their decisions, the teams continue making decisions for the following year, considering the previous results. The game lasts a couple of years until the results obtained by each team distinctively show the success or failure of their strategies.

### 2.1. THE DECISIONS

The management team of every company is responsible for making a series of decisions. These decisions concern production, labour, plant operation, marketing and financial

aspects. In making all these decisions, many different environment factors must be considered. The following sections describe each of these decisions briefly with the relevant environmental factors and with the expected results of these decisions. The relations between the environmental factors and the decisions were deduced from the BSG software manual (Thompson and Stappenbeck, 1995).

### 2.1.1. PRODUCTION DECISIONS

- Pairs to be manufactured

  The number of pairs in inventory that are unsold in the previous year, the expected demand in the market for the coming year, the reject rate estimation and the sale amount estimation are the factors that need to be considered.

- Percent use of long-wear material

  According to the quality that is targeted for the products, the percentage of long-wear material should be decided. More use of long-wear material will result in higher quality, but more expensive products.

- Number of models

  Three choices are defined for the number of models: 50, 100 and 200. If the number of models is higher, the products will appeal to a wider range of customers and as a result more will be soid. However, the cost of production will be higher because of switching between the production of different models. The number of models is also related to the quality of the products. When the number of models is small then the expertise of the workers increases and this results in high quality products. However, when the number of models is high, the workers do not work on the same model long enough to have expertise, and the quality of the products tends to be lower.

- Quality control budget & Styling/Features budget

  The amounts spent on the quality control and styling/features budgets. These amounts are directly effective in the quality of the products in the coming year. The more money spent per pair, the higher is the quality.

- Methods improvements budget

  The amount spent on the improvement of the plant efficiency. The money spent to improve the plant efficiency acts to reduce the supervision costs as much as 25%, reduce production run set-up costs by as much as 50% and increase the annual

productivity of the workers by as much as 10%. The improvements that are the results of this expenditure depend on the amount spent per pair. The more money spent per pair, the higher the improvements.

### 2.1.2. LABOUR DECISIONS

- Number of workers employed

  The number of pairs that will be produced and the worker productivity estimation are the two factors that need to be considered for this decision.

- Annual wage & Incentive pay per pair

  Annual wage per worker and incentive pay per pair are effective in worker productivity. While a raise boosts the productivity, a cut decreases it substantially. The worker productivity increases decreasingly as the incentive pay increases. The increase in the worker productivity does not increase the plant capacity, rather it decreases the number of workers that need to be employed. These decisions play also an important role in the quality of the products, because the more income the workers have, the more motivated they become and as a result the quality of the products increases.

### 2.1.3. PLANT DECISIONS

The results of the plant decisions do not become effective immediately in the year following the decisions, but in the next. Their payments are also made in the year they become effective. The following are the plant decisions:

- Automation options

  There are four different automation options (A, B, C and D). They have an increasing effects on the worker productivity and decreasing effects on the production run set-up costs by certain percentages. This is caused by the robotics-style technology installed in the plant as a result of automation. Every option has a predetermined cost that has an increasing effect on the cost of production. In the determination of the automation option, the increase in the market demand for products and the worker productivity play important roles. The automation option decision can be made once in the lifetime of the game, and the decision that is made stays valid until the end of the game.

- Expansion

  According to the increasing demand in the market, the production capacity of the plant can be expanded in order to satisfy the expected demand in the market. The increase in capacity is made by at least one million pairs per year with a certain cost. Although there is no limit on the amount a plant can be expanded, more than necessary expansion of a plant will only have an increasing effect on the production cost of the plant.

### 2.1.4. MARKETING DECISIONS

- Wholesale Price to dealers

  The wholesale price to dealers is the key determinant of the market share. However, a low price alone will not be enough for a company to become the market leader.

- Advertising Budget

  The amount of money spent on advertisement has an increasing effect in the product quantity that will be sold in the coming year. As well as the amount of money spent for the coming year, the cumulative amount spent until that year has also importance in that quantity.

- Number of retail dealers

  Retail dealers sell products directly to customers. Therefore, the more dealers there are, the more customers will be serviced and as a result the more products will be sold. There is no limit on the number of retail dealers that a company can have, but retail dealers also increase the cost of products because the company needs to support its dealers financially.

- Customer rebates

  There are four rebate options for the products: no rebate, $1 rebate, $3 rebate and $5 rebate. The customers intend to buy the products that offer more rebates. Therefore, the number of products that will be sold is expected to be more with more rebates offered.

- Number of customer service representatives

  One customer service representative can service fifty retail dealers. Having one service representative for less than fifty retail dealers is expected to increase the amount of products sold and having one service representative for more than fifty

retail dealers is expected to diminish the amount of products sold. In order to have a positive impact of this decision on the amount sold, the ratio should be considered rather than its absolute value.

- Delivery time

  Four different delivery time options are available: four, three, two weeks and one-week. A lower value has an increasing effect on the amount of product that will be sold.

#### 2.1.5. FINANCIAL DECISIONS

- Request for short-term loan

  A short-term loan comes from financial institutions in order to support the financial status of the company. Short-term borrowing means more money to spend on production, therefore it can indirectly affect the amount of products that will be sold in the coming year. Short-term loans need to be paid back in the following year with a predetermined interest. When a company ends up with a negative cash balance at the end of a year, it is automatically given a short-term loan to raise its balance to zero.

- Common stock shares issued

  A company can support itself financially by issuing new stock shares. Issuing new shares will decrease the price of the prevailing shares.

- Bond issue

  All bonds are issued for a ten-year term and they can be considered to be long-term loans. The money borrowed by issuing ten-year bonds is repaid annually in equal instalments with interest. The interest rate of each bond issued depends on the company's bond rating in the year it is issued. The number of bonds a company can issue is limited by ten, two of which have already been issued in previous years.

- Dividend declaration

  Annual dividend declaration is important for the stockholders and the stock price is directly proportional to the dividend declaration.

### 2.2. THE FINANCIAL INDICATORS

There are five financial indicators (revenues, profit, earnings per share, return on equity and cash balance) that show the results of the decisions made by each team.

- Revenue

  Revenue is the total amount obtained as a result of selling products in the market. The multiplication of the total number of products sold in a year by the price per product gives the revenue, assuming that there is only one product.

- Net income

  Net income is the amount left from the revenue after deducting all the production costs, the depreciation costs, interest paid and taxes. This is the amount that the company earned in that year. A large revenue does not mean that the company is very successful, unless it also has a high percentage of net income.

- Earnings per share (EPS) & Return on equity (ROE)

  EPS and ROE values are the indication of the company's profit relative to the number of shares outstanding and to the total equity that is invested by the stockholders. EPS is expressed in $/share and it is calculated by dividing the annual net income of the company by the total number of outstanding shares. ROE is equal to the percentage of the annual profit with respect to the total equity of the stockholders. At the beginning of the game, the EPS value is very close to $1.5 and ROE value is around 15%. An increase in these indicators will please the company's shareholders and as a result the share price of the company will increase, with a positive impact on the reputation of the company. Since the goal is to please the shareholders, the management teams should set their goal in order not to decrease the values of these indicators too much. These are relative indicators, therefore they show the success of the management teams more realistically than the other indicators.

- Cash

  This is the amount not spent at the end of the year. If the estimated expenses for the coming year are less than the actual expenses, the company will have a negative cash balance and will be obliged to finance its expenses by short-term loans. A cash balance that is close to zero without an obligation to borrow money at the end of the year means that the company uses all of its financial resources efficiently and makes good estimates.

# 3. OVERVIEW OF THE NEGOPLAN SYSTEM

This chapter is an introduction to the Negoplan system, which gives the reader background knowledge to understand the extensions and modifications implemented in the system and described in the following chapters.

Section 3.1 describes the modelling methodology on which Negoplan is based and the class of problems addressed by Negoplan according to this methodology. The Negoplan constructs used in the representation of these problems and closely related to our extensions of the system will be introduced in Section 3.2, accompanied by examples. Negoplan's inference engine will be demonstrated in Section 3.3 and the limitations of the present Negoplan system for some problems will be discussed in Section 3.4.

## 3.1. PROBLEMS ADDRESSED BY NEGOPLAN

Negoplan is a domain-independent decision support system which can be customised for different application domains. Negoplan is based on the *restructurable modelling methodology* (Kersten *et al.*, 1991) which enables the representation, simulation and support of sequential, context-dependent decision-making problems. Making a decision is often associated with the choice of one alternative from a given set in order to achieve certain goals.

In sequential decision making problems, in order to achieve a principal goal, a sequence of decisions needs to be made. Each decision made in the sequence is a contribution towards the achievement of the principal goal and each decision highly depends on the earlier decisions made. For instance, in a representation of a manufacturing company the necessary decisions might include determining the quality and the amount of the products, selecting the marketing strategy and so on. The target quality of the products and the strategy that will be followed in marketing these products may be effective in the determination of the amount that should be produced.

Context-dependent decision making problems are mutually related with the agents who play a role in them. Context dependent decisions may vary depending on the state or behaviour of their participating agents. A decision made may affect the agents and cause their reactions. It is usually necessary to adjust context-dependent decisions after considering the reactions of their participating agents (Szpakowicz *et al.*, 1990). For instance, a decision to produce low-quality products while the market tendency is towards high-quality products may lead to a high-investment decision in advertisement in order to

compensate for the wrong quality decision and achieve the target sale quantity in the upcoming year.

The decision context—the participants and the situation—may be disregarded if the decision is considered as a single act. If, however, there is a sequence of decisions, a partial decision usually modifies the context in which subsequent decisions are made.

The Negoplan system permits the simulation of the interaction between two agents and their decision environment. The organisational structure of Negoplan (Kersten, 1994) comprises the supported agent (who may also be the user of the system), the opposing agent (opponent) and the decision environment; this terminology reflects the original, and later extended, purpose of Negoplan as a tool for the modelling of two-party negotiations. The environment refers to the circumstances under which the interactions occur rather than the physical environment. Since the opponent is directly related to the supported agent's current and future decisions, it is considered separately from the decision environment.

A decision-making problem is viewed from the supported agent's point of view and the goals or the internal state of that agent is represented by the system with a hierarchical graph that is called a problem decomposition graph (PDG). Throughout the simulation of the decision-making problem, the PDG of the supported agent is modified in response to the changes occurring in the states of the two agents and in their decision environment. These modifications represent the context-dependent variations in the goals or in the internal state of the supported agent.

The application of Negoplan for a decision-making problem simulation starts with the definition of the supported agent's PDG and continues with the definition of the behaviour of the supported agent in different situations, the opponent and the environment that are participating in this decision-making problem. The representation and the modifications of the PDG are expressed in a rule-based formalism (Kersten and Szpakowicz, 1990). The behaviour is defined by metarules, using a declarative language embedded in the system. Since Negoplan is implemented in Prolog, this declarative language shares syntax with some of the Prolog's primitives. The simulation of the problem proceeds by using logical inference, causing the agents' behaviour to occur according to the state of the problem simulation. The realisation of this behaviour modifies the state of the problem and the PDG of the supported agent. The state of the problem is determined by a set of facts that introduce the states of the agents participating in the problem.

The medical decision-making problem simulation (Kersten *et al.*, 1994b) is one of the larger applications of Negoplan. This problem involves a (simulated) patient and a medical student who is trying to make a correct diagnosis and select an appropriate treatment for the patient's illness. The problem is viewed from the patient's point of view and the medical status of the patient is depicted in a PDG. The initial graph introduces all the visible medical symptoms. The task of the medical student is to reveal the illness of the patient by following a correct sequence of medical decisions and to come up with a correct treatment for this illness. Some possible actions of the medical student are medical examination, application of medical tests, learning the history of the patient and an interim treatment. Each of these actions will have a modifying affect on the patient's PDG, but will lead to an expected PDG only if correct actions are taken and they are taken in a certain order. For instance, the interim treatment decision taken before medical examination, before learning the history of the patient and before the application of medical tests may lead to an unexpected PDG introducing negative results in the medical status of the patient. However, the interim treatment decision taken after those decisions and considering the facts that are collected from the results of those decisions, may lead to a PDG introducing improved visible symptoms in the medical status of the patient.

## 3.2. NEGOPLAN CONSTRUCTS

A Negoplan application comprises four major components: a rule base, a metabase, a section containing application-specific Prolog definitions and a customisation section. The rule base contains the PDG which is a description of the problem in the form of a decomposition hierarchy of rules. The metabase contains metafacts that characterise the position of each agent and the metarules that are used to simulate the dynamics of the decision problem (Noronha and Szpakowicz, 1996). The changes in the positions of the agents may require modifications in the PDG and this is achieved by restructuring metarules.

Negoplan constructs that are introduced in this section are only the ones that are related to the extensions and used in the "Business Strategy Game" application. This section is partially quoted from the Negoplan Case Author's Manual and the examples are taken from the case study for the medical decision-making problem (Kersten *et al.*, 1994).

A Negoplan application is mainly accomplished by customising the Negoplan's default appearance and behaviour attributes to suit a given decision-making problem, by defining the PDG of the supported agent and by writing metarules that determine the behaviour of the agents participating in the problem in various situations.

In the discussion of the Negoplan constructs, the terms that appear in *italics* will represent the name of a construct, whereas a term in `typewriter font` will represent a text that is typed literally. For instance, in the definition of the supported agent: `us(`*supported agent*`)`, `us` is typed literally, but *supported agent* is a placeholder for the actual name of the agent.

### 3.2.1. PROBLEM DECOMPOSITION GRAPH

In a Negoplan application, the hierarchy of goals or the internal hierarchical organisation of the supported agent is represented by the problem decomposition graph (Kersten and Szpakowicz, 1994b) in which a node is described by a decomposition rule that relates a goal to subgoals or an element of the agent's structure to substructures. A decomposition rule is in the form of:

$f_0$ <- $f_1$ & ... & $f_n$.

where $f_0$ denotes a goal or a structural element and $f_1$, ..., $f_n$ are its top-level components. An element that is not decomposed any more is called a fact.

Decomposition rules can be interpreted as implication (Kersten and Szpakowicz, 1990). For the left-hand side of a rule to be present or achieved, all the right hand side components should be present or achieved. The symbol & represents the logical conjunction. For instance the "presence of the general exam follows from the presence of the facts 40° fever and irregular pulse and normal head size and normal blood pressure".

Each element ( $f_i$ ) on the right-hand side of a decomposition rule is written as a predicate in Prolog and is in the form of:

*name*$(p_1, \ldots, p_n)$.

where *name* indicates the nature of the element and the parameters $p_1$, ..., $p_n$ represent details, for example, `blood_pressure(normal)`.

The parameters of a predicate are usually either constants or variables. In Prolog, constants are numbers or symbolic constants which are identifiers beginning with a lowercase letter (`patient`) or identifiers that are quoted (`'Sick Child'`) and variables are identifiers beginning with an uppercase letter.

The symbol <- in an individual decomposition rule can be interpreted as an implication (Kersten and Szpakowicz, 1990), as a consequence or as a structural decomposition. The symbol & represents the logical conjunction. For instance, this rule:

```
general_exam  <-  fever(40) &
                  pulse(irregular) &
                  head_size(normal) &
                  blood_pressure(normal).
```

can be described in several ways depending on the interpretation of the <- symbol. "the presence of the facts *40° fever* and *irregular pulse* and *normal head size* and *normal blood pressure* implies the presence of the *general exam*" when it is interpreted as an implication, "the presence of the facts *40° fever* and *irregular pulse* and *normal head size* and *normal blood pressure* are the consequences of the *general exam*" when it is interpreted as a consequence and "a general exam consists of checking the fever, the pulse, head size and the blood pressure" when it is interpreted as a structural decomposition.

Negoplan does not force the reader to interpret the rules in a certain way. Therefore, in order to interpret a PDG, it is necessary to know the way the problem is understood by its designer.

The initial internal organisation of the medical decision-making problem represents the medical state of a `sick_child`. One of the initial PDG depicts the medical state of a `sick_child` who has a complaint in his `intestinal_system` caused by a `virus` with the `visible_symptoms` of `diarrhea`, `refusal_to_eat`, `neutral irritability` and `vomiting`.

```
sick_child            <-   intestinal_system.
intestinal_system     <-   virus.
virus                 <-   visible_symptoms.
visible_symptoms      <-   diarrhea &
                           refusal_to_eat &
                           irritability(neutral) &
                           vomiting.
```

Throughout the simulation of the problem, this organisation expands as a result of the decisions taken by the medical student. The expansion that lets the student reveal the illness of the `sick_child` indicates that his decisions are correct. One of such final organisations of the medical decision-making problem is as follows:

```
sick_child            <-   intestinal_system.
intestinal_system     <-   virus.
virus                 <-   visible_symptoms &
```

```
                              general_exam &
                              patient_history.
visible_symptoms        <-    diarrhea & refusal_to_eat &
                              irritability(neutral) & vomiting.
general_exam            <-    fever(40) & pulse(irregular) &
                              head_size(normal) &
                              blood_pressure(normal).
patient_history         <-    diet(no_change) &
                              loss_of_weight(0.4) &
                              family(no_similar_symptoms) &
                              bowel_movements(normal).
```

### 3.2.2. METAFACTS

A predicate with a logical value true or false assigned to it is called a metafact. The logical value assigned to a metafact reflects the existence or non-existence of that predicate at a certain time of the simulation. With the use of side names, metafacts are ascribed to specific agents involved in the decision-making problem, allowing the statement represented by the predicate to be viewed, perhaps differently, by different agents. Metafacts are written as follows :

*side* : *predicate*(...) ::= *logical_value*

For instance, the following two metafacts have been taken from the medical decision-making problem:

```
sick_child : 'refusal to eat' ::= true
student : 'interim treatment' ::= false
```

The former reflects the `sick_child`'s `refusal to eat` and the latter the absence of an `interim treatment` request by the `student` at a certain time during the simulation of the problem.

The set of metafacts that are available at a certain time of the simulation represents the current state of the problem.

### 3.2.3. METARULES

The behaviour of the agents in different situations and the interactions between the agents is defined by means of metarules. A metarule can be viewed as a cause-effect relation

consisting of metafact conjunctions and special constructs. The general form of a metarule is as follows:

*triggers* ==> *conclusions*.

The following is a fictitious metarule describing the `hospitalize` action taken by the medical `student` if the `fever` of the `sick_child` is above 40°.

```
sick_child : fever(Temperature) ::= true &
{ Temperature > 40 }
==>
student : hospitalize ::= true.
```

A metarule is triggered only if the metafacts that are listed on its left-hand side have been encountered in the set of metafacts at some time during the simulation. When triggered, it results with the effects listed on its right hand-side causing some changes in the state of the decision-making problem.

The predicates in the metafacts on the left-hand side of a metarule may have variable parameters. These will be instantiated by its corresponding metafact when it is encountered in the set of metafacts. A parameter can also be affected by tests or calculations performed by embedded Prolog calls. Such a call is written in curly brackets at the same position in a metarule that a metafact would occupy. An embedded call may check a condition related to a parameter or may compute its value. If an embedded call fails, it prevents the application of the metarule in the current situation.

There are four kinds of metarules with different functionalities.

#### 3.2.3.1. Response metarules

Response metarules describe the reactions of the agents participating to the problem that will appear in various situations. The *triggers* are conjunctions of metafacts and embedded calls.

*triggers* ==> $metafact_1$ & $metafact_2$ & . . . & $metafact_n$.

When a response metarule is triggered, the metafacts listed on its right-hand side are integrated with the set of metafacts existing in the system.

The following is a fictitious response metarule describing the change in the medical condition of the `sick_child` having a `fever` above 40°, when the medical `student` has decided to apply `interim treatment`. Once this metarule has been triggered, the (simulated) `fever` of the `sick_child` drops by 2°.

```
sick_child : fever(Temperature) ::= true &
{ Temperature > 40 } &
student : 'interim treatment' ::= true
==>
{ New_Temperature is Temperature - 2 } &
sick_child : fever(New_Temperature) ::= true &
sick_child : fever(Temperature) ::= false.
```

### 3.2.3.2. Selection metarules

When Negoplan is used as a decision support or simulation tool, it may be necessary to have user interaction in order to assist the decisions of an agent participating to the decision-making problem. User assistance can be provided to the decisions of any agent and this functionality makes Negoplan very suitable for training or evaluating the complex decision making skills of its users.

*triggers* `==> select` $(choice_1, ..., choice_k)$ `&` *other conclusions*.

When a selection metarule is triggered, the control is given to the user to let him tailor the right hand side of the metarule according to his objectives. The choices that are selected by the user are transformed into metafacts with the side name of the agent that is introduced by the `selection_owner` predicate and those metafacts are integrated with the set of metafacts existing in the system.

The following is a fictitious selection metarule which is triggered when the `sick_child` has `diarrhea`. The user is asked to select one or more among the list of possible actions. Since in the medical decision-making problem simulation the `student` is decided to be the selection owner, the selections of the user are transformed into metafacts with the side name of the `student` and integrated with the set of metafacts existing in the system.

```
sick_child : diarrhea ::= true
==>
select (    'contact specialist',
            'general exam',
            hospitalization,
            'patient history' )
with_message 'Choose initial actions to perform:'.
```

3.2.3.3. Restructuring metarules

As a result of the decisions made and actions performed by the agents participating in the decision-making problem, the previously defined PDG may become inadequate for the current state of the problem and the supported agent may need to restructure its hierarchy of goals or its internal hierarchical organisation considering the changes that have occurred in the state of the problem.

The transformation of the PDG is performed by the restructuring metarules, which are in the form of:

*triggers* `==> modify (`$rule_1, ..., rule_k$`) .`

The rules that are given in the right hand side of the restructuring metarules either replace or extend the existing decomposition rules that appear in the PDG. If the rule that appears in a restructuring metarule is defined with the `<-` operator, this rule replaces a rule with the same left-hand side in the current PDG. However, if the rule is defined with the `<-&` operator, a rule with the same left-hand side must exist in the current PDG and it is extended rather than replaced. For instance, an existing decomposition rule $r_0$ `<-` $r_1$ `&` $r_2$ will be replaced with the rule $r_0$ `<-` $r_3$ `&` $r_4$, if this rule appears in a triggered restructuring metarule. However, the same decomposition rule will be extended by the rule $r_0$ `<-&` $r_3$ `&` $r_4$ resulting in $r_0$ `<-` $r_1$ `&` $r_2$ `&` $r_3$ `&` $r_4$ if this rule appears in a triggered restructuring metarule.

The following fictitious metarule describes the restructuring of the PDG in response to the general exam decision in the medical application. As a result of this metarule, the part of the PDG of the sick child depicting visual symptoms related to an intestinal virus is extended by the `general exam` rule. If there already is a rule that defines `general exam`, it is replaced, if there is not, a new rule is added to the PDG of the `sick_child`.

```
student : 'general exam' ::= true
==>
modify (    virus( int ) <-& general_exam,
            general_exam <-   age( '13 months', 1 ) &
                              'blood pressure'( low, 1 ) &
                              pulse( high, 1 ) ).
```

A PDG can also be restructured piece by piece, using the `procedure` construct in response metarules. This mechanism makes it possible to save a change to a PDG for further use; that is, a virtual change can be made in advance in a response metarule. When a restructuring metarule is triggered, all the partial changes that have been saved earlier

and the changes introduced in that metarule can be gathered together by calling the `save_all_modifications` predicate. For instance, the first part of the change introduced in the restructuring metarule shown above can be made in advance by the following right-hand side of a fictitious response metarule:

```
...
==>
student : procedure ( virus( int ) <-& general_exam ) ::= true.
```

This change can be combined with the other changes by calling the `save_all_modifications` predicate in the following fictitious restructuring metarule:

```
student : 'general exam' ::= true &
{ save_all_modifications(student) }
==>
modify (    general_exam <-   age( '13 months', 1 ) &
                              'blood pressure'( low, 1 ) &
                              pulse( high, 1 ) ).
```

#### 3.2.3.4. Termination metarules

When a certain condition is observed during the simulation, one of the termination metarules, which usually displays the cause of termination corresponding to that condition, is triggered and the decision-making process is terminated.

*triggers* `==> terminate` *causes of termination*.

The following termination metarule is triggered in response to the `student`'s incorrect `hospitalisation` decision. When triggered, the simulation of the problem terminates by displaying the text provided as the cause of termination.

```
student : hospitalisation(_) ::= true
==>
terminate
'Condition of the patient does not warrant hospitalisation.'.
```

### 3.2.4. PACKETS

Packets are used to organise metarules into modules such that at most two modules may be active at any given time. A metarule can only belong to one packet; it is labelled with the name of the packet as follows:

*metarule* `---` *packetname*.

The following metarule is a fictitious response metarule which belongs to the packet called `important_stage` in which all the crucial decisions are made.

```
sick_child : fever(Temperature) ::= true &
{ Temperature > 40 }
==>
student : hospitalize ::= true
--- important_stage.
```

The metarules that are not labelled with any packet name belong to the global packet which is always active during the simulation of the problem with a named packet. At any given time, only the metarules that belong to the active packets may be triggered. Switching to another packet is accomplished with the `switch_to` command, which can appear only in modification metarules:

*modification_metarule* `switch_to` *packetname1* `---` *packetname2*.

Packets can be used to represent different phases of the decision-making problem.

### 3.2.5. Implementing an Application

Implementing an application of the Negoplan system requires the determination of the agents that will be involved in the decision-making problem and the introduction of those agents' names into the system by means of the following Prolog predicates:

`us(`*supported agent*`).`
`them(`*opponent agent*`).`
`neutral(`*environment*`).`

For instance, the names of the agents that participate in the medical decision-making problem are introduced with the following predicates:

```
us(sick_child).
them(student).
neutral(environment).
```

In the rest of the implementation, these names are used to differentiate the facts that belong to different sides.

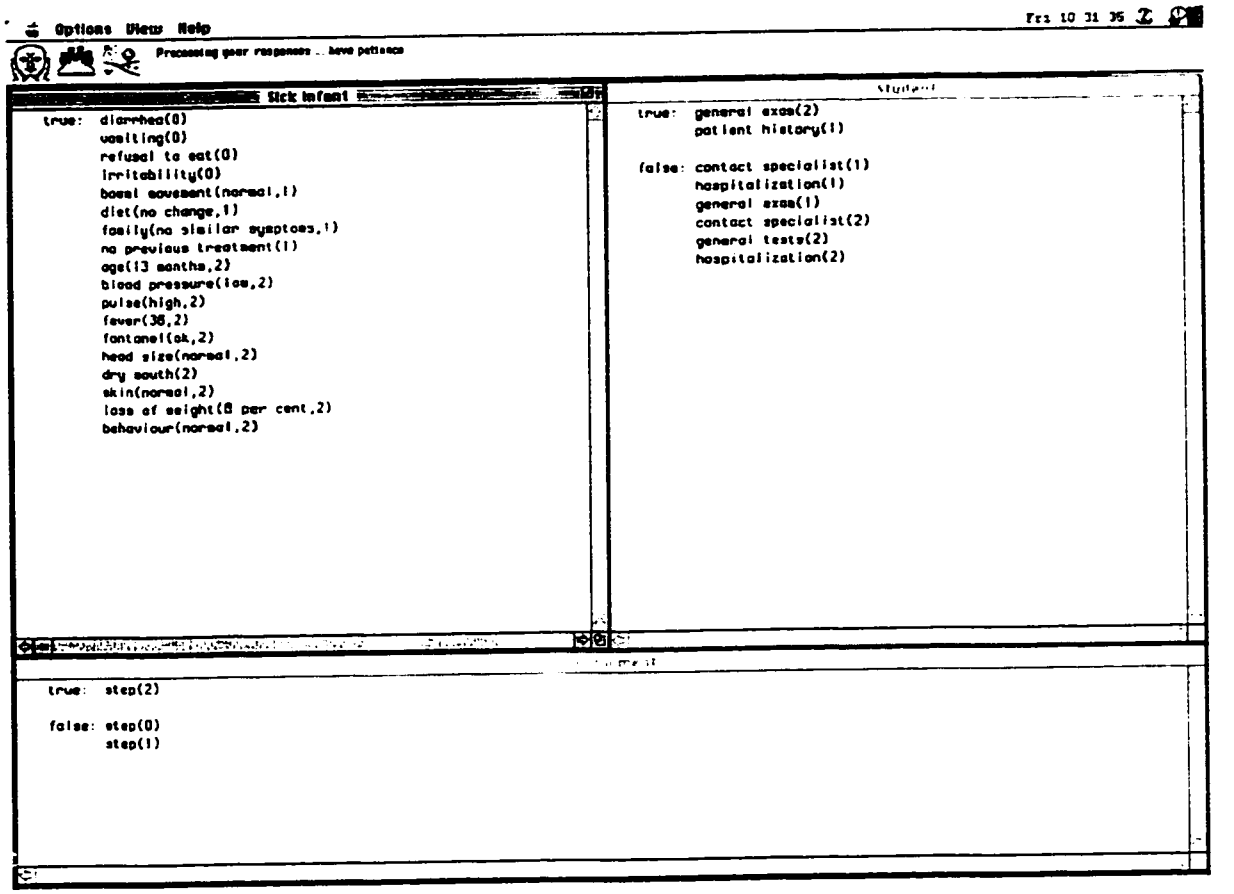



Figure 1. A snapshot from the medical decision-making problem simulation

The Negoplan user interface on Macintosh systems provides a different window for each agent and the facts belonging to different agents are displayed in their corresponding windows. In order to identify which agent's metafacts are displayed in which window, these windows are given a name with the following Prolog predicates:

`side_window( 'User',` *supported agent*`).`

`side_window( 'Other',` *opponent agent*`).`

`side_window( 'Environment',` *environment*`).`

For instance, in the medical decision-making problem simulation, the windows are given the following names:

```
side_window( 'User', 'Sick infant').
side_window( 'Other', 'Student').
side_window( 'Environment', 'Environment').
```

The names of the agents that are defined to identify their metafacts and the names of the windows that are defined to identify which agent they represent do not need to be the same. In fact, the former names are only observable by the person who is customising the system and the latter names are only observable by the person who is using the application.

The user interface for the medical decision-making problem simulation is introduced in Figure 1. The metafacts of the agents, `sick_child`, `student` and `environment`, are displayed in their corresponding windows that are named as `'Sick infant'`, `'Student'` and `'Environment'`. The figure depicts the time after when the medical `student` requested the `patient history` and general exam. These actions are displayed in the `student`'s window and new symptoms that are discovered as a result of these actions are displayed in the `sick_child`'s window.

## 3.3. NEGOPLAN INFERENCE ENGINE

The execution of a Negoplan application for a decision-making problem starts by loading the initial PDG of the supported agent and the metarules defining the behaviour of the agents participating in the problem and the interactions between them. The system can also be given metafacts that define the initial states of the opponent and the environment.

A simulation starts with the generation of the metafacts belonging to the supported agent from the initial PDG by using goal-directed (backward) reasoning and proceeds with forward chaining of response, selection and termination metarules (the response cycle), forward chaining of modification metarules (the restructuring cycle) and backward reasoning on the modified PDG.

### 3.3.1. GOAL-DIRECTED (BACKWARD) REASONING

The predicates at the leaves of the PDG are identified and assigned a logical value `true` or `false` in such a way that the topmost element of the PDG becomes `true` when backward reasoning is applied. In a PDG, if disjunctions are used, there may be more than one alternatives which makes the topmost element of the PDG true. At the time of backward reasoning, at each disjunction node only one subtree is arbitrarily chosen and the predicates that are at leaves of that subtree are assigned a logical value. The predicates that appear in the subtrees that have not been chosen are not assigned any value. Those predicates that are assigned a value are labelled with the supported agent's name in order to constitute its initial set of metafacts. Backward reasoning is applied to the graph at the

beginning of the simulation and also every time it is modified after the restructuring cycle in order to refresh the representation of the state of the supported agent.

### 3.3.2. RESPONSE AND RESTRUCTURING CYCLES

The response cycle can be interpreted to be the period when all the participating agents including the environment respond to the situation. The response of the agents can be the result of an interaction between the supported agent and the opponent or just a reaction to a change in the environment. The behaviour of the agents that arises during the response cycle is defined by the response, selection and termination metarules.

In the restructuring cycle the supported agent reconsiders his current state and his objectives according to the changes that occurred in the problem state in the response cycle. As a result of the restructuring cycle modifications may occur in the PDG, and the decision-making problem may also switch into another phase. This behaviour of the supported agent is defined by modification metarules.

Although more than one response metarule may be triggered in the response cycle, only one modification metarule can be triggered in the restructuring cycle. Whenever a response metarule is triggered, the metafacts listed on its right hand side are added to the existing metafacts. When a restructuring metarule is triggered, the rules that are listed on its right hand side are incorporated into the PDG of the supported agent. In the restructuring cycle only one metarule can be triggered, because the objectives of the supported agent are supposed to be enhanced only in one way. If there is a packet-switch clause in the modification metarule, the metarules for the current phase are disabled and those for the new phase are made available for chaining.

The response cycle differs from the restructuring cycle only in their conclusions. The metarules that are triggered in both of the cycles are determined by the same forward chaining mechanism implemented in the Negoplan system (Figure 2). At the beginning of both cycles the existing metafacts of the opponent and the environment are added to the supported agent's metafacts that are extracted from its PDG. The metafacts, one by one in an unspecified order, are matched with the metafacts that appear in the left-hand side of the metarules in the active packets. The metarules whose left-hand sides contain a metafact that matches an existing metafact, may be triggered and are placed in a special data structure called agenda after the matched metafact is removed from their left hand sides. If the left-hand side of a metarule that is already on the agenda contains a metafact that matches an existing metafact, that metafact is simply removed from its left-hand side. Embedded Prolog calls that appear on the left-hand side of the metarules on the agenda

execute when all the metafacts preceding it have been removed. When all the metafacts are removed and all the embedded Prolog calls execute successfully on the left-hand side of the metarule, that metarule is triggered and its conclusions are performed.

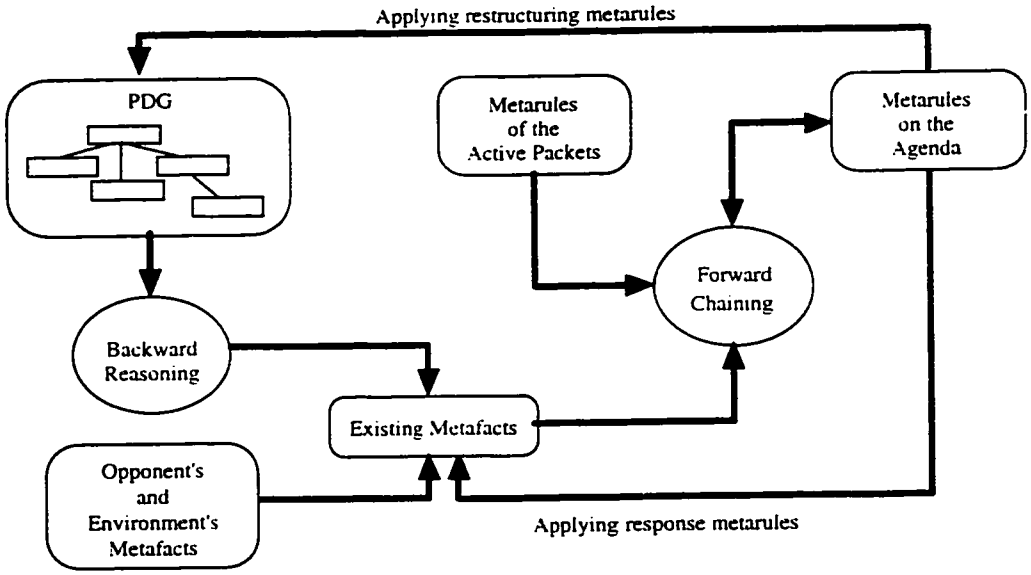



Figure 2. The Forward Chaining Mechanism of the Negoplan System

### 3.3.3. METARULE INDEXING MECHANISM

The forward chaining mechanism examines a metafact at a time and determines the metarules that belong to the current packets and contain a metafact matching the examined metafact on their left-hand sides. The forward chaining mechanism later removes the matching metafact from the left-hand sides of those metarules and places those metarules on the agenda giving them a chance to be triggered. If a metarule that is already on the agenda contains a metafact matching the examined metafact, the forward chaining mechanism simply removes that matching metafact.

In order to find the metarules within the current packets that contain the examined metafact, the metarules are indexed at the beginning of the simulation according to the packets to which they belong and according to the metafacts that they have in their left-hand sides. The index entries are stored in `rule_index` clauses and have the following structure :

`rule_index(`*metafact, list of (packet, response metarules, modification metarules)*`)`

For instance:

```
rule_index(metafact1,[  [packet1,[1,2,3],[4,7,11]],
                        [packet2,[15,16],[21]] ]).
```

When, for example, the forward chaining mechanism examines `metafact1` and the current packet is packet1, the response metarules 1, 2 and 3 in the response cycle and the metarules 4, 7 and 11 in the restructuring cycle will be placed on the agenda.

Indexing of the metarules is performed by the metarule reader which parses the metarules at the beginning of the simulation in order to determine which metafacts appear in their left-hand sides. For each metafact that appears in the left hand side of a metarule and that does not have an index entry, a new entry is added. In that entry, a new *(packet, response metarules, modification metarules)* triple is constructed. The name of the packet that the metarule belongs to is written in the *packet* field and the metarule number is appended to the appropriate list of metarule numbers according to the type of that metarule. For a metafact that appears in the left-hand side of a metarule and has an index entry, only the metarule number is appended to the corresponding list of the triple which has the packet where the metarule occurs. If there is already an index for the metafact, but not a *(packet, response metarules, modification metarules)* triple which has the packet in which the metarule occurs, a new triple is constructed and added to the list of triples.

## 3.4. LIMITATIONS OF THE PRESENT NEGOPLAN SYSTEM

The present Negoplan system has weaknesses that impose restrictions on the supported agent and the problem domain.

The supported agent cannot learn new concepts (Kersten and Szpakowicz, 1994). The behaviour of all agents participating in a decision-making problem is described when an application is implemented and it can not be modified throughout the execution of the application. In real life, throughout the decision-making processes the participants can find reasons behind actions, goals and plans of other participants and as a result learn new concepts. However, the supported agent can only determine and verify its own goals and plans by considering the reactions of the others.

Another limitation of the Negoplan system is that it provides the constructs to associate metafacts with three different entities: the supported agent, the opponent and the environment. This creates a limitation on the number of agents that can be represented in a decision-making problem simulation and in the definition of the metarules that represent the behaviour of the agents in different situations. As a result of this limitation, only the decision-making problems that involve two agents and their decision-making

environment can be simulated by the Negoplan system. Previous applications of Negoplan include a medical decision-making problem, a robot action planning, a car buying decision-making problem and a labour-management negotiation. The two agents in these applications are, respectively, the medical student and the sick child, the robot and the terrain, the car dealer and the customer, the trade union and the management team of the company.

Limiting a decision-making problem simulation to at most two agents actively participating in the decision-making process may not be very realistic for many decision problems. For instance, in the car buying application only two agents, the dealer and the customer, are represented by the system, which is not likely to happen in real life. There may possibly be more than one dealer who will have an effect on the decision of the customer. Therefore for some problems, there may be a need for representing more than one opponent and their behaviour. As well as having similar roles in the decision-making process, the opponents can also have very different roles from each other. For instance, in the car buying problem two used-car dealers who want to sell their cars to the customer would be expected to behave similarly under the same situations. However, the behaviour of a private seller who wants to sell his car to the customer would differ from the behaviour of the used-car dealers. As a result of this aspect of the decision-making problems, there is also a need for grouping together the opponents that are expected to behave similarly, in order to be able to represent their behaviour with the same metarules.

Chapter 4 describes how the limitation to two agents and the environment has been relaxed. This limitation has been selected in order to create a realistic decision-making environment for a managerial decision-making problem involving more than two agents.

# 4. EXTENSIONS OF THE NEGOPLAN SYSTEM

This chapter introduces the extensions that enable the application of the Negoplan system for decision-making problems that involve more than one opponent. All the necessary extensions have been implemented without changing the current meaning of the existing Negoplan constructs. Therefore, all the earlier decision-making problem applications that involve only one opponent can be run in the extended Negoplan system without any change.

The extensions of the Negoplan system are divided into three categories. The first category includes new constructs and changes in the meaning of some existing constructs. These changes enable the application of Negoplan for a broader class of decision-making problems. The second category includes modifications in indexing and forward chaining mechanisms. The third category includes changes that improve the system's performance—it is expected to decrease for new, more complicated decision-making problem applications.

In order to test the extensions, a Negoplan application has been implemented for a small decision-making problem that uses almost all of the new features. In this application a company (**seller**) wants to sell its products and a few other companies (**buyers**) make offers to buy those products. The goal of the seller is to select the most financially advantageous offers and to sell its products to the companies making those offers. Whenever possible, the examples in the following sections are taken from this Negoplan application. Its details will be presented after all the modifications related to the extensions have been introduced.

## 4.1. NEW AND MODIFIED NEGOPLAN SYSTEM CONSTRUCTS

In order to enable the application of Negoplan system to be customised for decision-making problems that involve more than one opponent, the meaning of some of its existing constructs has been extended and some new constructs have been defined. These changes are presented in the following two subsections.

### 4.1.1. CHANGES IN THE EXISTING CONSTRUCTS

While the meaning of some existing constructs is now extended, their previous meaning has been preserved so that all the earlier applications of Negoplan system can be run by the extended system without any change.

In the Negoplan system without the extensions, the name of the opposing agent that participates in the decision-making problem is introduced by the following Prolog predicate:

`them(`*opponent name*`).`

where *opponent name* can only be a constant. In order to allow more than one opponent (necessary in the new type of problems), the meaning of the `them` predicate has been extended by allowing a list of names as well:

`them([`*opponent name*$_1$, .... *opponent name*$_N$`]).`

where each *opponent name*$_i$ is a constant.

For example, the following Prolog predicate introduces five opponents in the seller-buyers decision-making problem.

```
them([ buyer1, buyer2, buyer3, buyer4, buyer5 ]).
```

When the `them` predicate has only one name as argument, the Negoplan system understands that the decision-making problem involves only one opponent. When it has a list of names as argument, the problem is understood to involve as many opponents as there are elements on list. Therefore, the argument of the `them` predicate has an important role in the determination of the number of opponents participating in the problem and it is used as the main indicator in the rest of the simulation whenever the number of opponents makes a difference.

### 4.1.2. NEW CONSTRUCTS

This section introduces new constructs that will enable the grouping of the opponents with similar behaviour and that will be used to refer to those opponents in metarules defining their behaviour.

#### 4.1.2.1. Groups, group members and generalised metarules

Some decision-making problems may involve not only opponents with different behaviour, but also opponents that respond similarly to the same situation. Therefore, some problems may be best implemented if it is possible to group the opponents that behave similarly. As a result of this ability, generalised metarules can describe common behaviour of these opponents, instead of having a copy of that metarule once for each of these opponents. From this perspective, in extended Negoplan, a group is defined as a collection of opponents with a common behaviour, a group member is defined as an

opponent that belongs to a group, and a generalised metarule is defined as a metarule that describes the common behaviour of group members.

#### 4.1.2.1.1. Comparison with the Object-Oriented Design

The relation between a group, its members and their generalised metarules in extended Negoplan can be conceptually compared with the relation between a class, its instances and their methods in object-oriented design (OOD). In OOD, a class uses methods to specify the behaviour of the objects that are its instances (Eliëns, 1995). This resembles Negoplan with extensions where a group uses generalised metarules to specify the behaviour of its members.

One difference between these concepts is that Negoplan extensions do not prevent group members from having personal behaviour. However, in OOD two instances of the same class can only be manipulated by the methods defined for that class, unless that class is a superclass and those instances are objects of different subclasses of that superclass with additional methods.

Another difference is that Negoplan extensions provide only one level of abstraction while OOD can provide multi-level abstraction. Negoplan allows us to create more than one group in an application but groups cannot be interrelated, unlike classes which can be derived from other classes in OOD. Therefore, inheritance which is one of the most important features of OOD is provided by these new extensions of Negoplan only to a certain extent: between the groups and their members.

#### 4.1.2.1.2. Implementation details

The grouping is presented to the system with a new predicate `group_members`, placed in a case customisation file. The form is as follows:

`group_members(` *group name*, `[` *opponent name*$_1$, ..., *opponent name*$_N$ `]` `)`.

where *group name* should be replaced with a name that represents all the opponents behaving similarly and each *opponent name*$_i$ with an actual name of an opponent belonging to the group. The number of opponents that a group can contain is restricted by the number of opponents defined in the problem. There may be many groups, but every opponent can appear in only one group. Some opponents might not be members of any group.

For example, the following predicates are taken from the seller-buyers application of Negoplan.

```
group_members( buyersA, [ buyer1, buyer2 ] ).
group_members( buyersB, [ buyer3, buyer4, buyer5] ).
```

These predicates represent two opponent groups with possibly different behaviour, with two buyers and three buyers respectively. Grouping means that the behaviour of the same group members is almost identical. There can also be some situations in which the same group members behave differently. Grouping can be advantageous only if the similar behaviour is more than the different behaviour of the opponents. Being in two different groups does not restrict the opponents to have similar behaviour in some situations. However, if there are many of such situations, it could be better to group those opponents.

This device not only helps avoid writing the same metarule once for each opponent that is a member of a group. Grouping the opponents can also play an important role in the maintenance of an application. For instance, the number of opponents in a group can be changed by simply changing the number of elements in that group's definition, without the need of updating the metarules that define their behaviour. As a result of this convenience, an application can easily be tested with different numbers of opponents and the user of the system can gain a better understanding of the problem. Also, modification of a common behaviour of group members can be achieved by the modification of the generalised metarule.

Generalisation of a metarule describing a behaviour common to a group of opponents could be achieved by using the group name instead of a single opponent's name wherever a member of that group is to be referred to. The generalised metarule can be interpreted as ready to be triggered once for each group member satisfying the left-hand side of the metarule. However, the use of the group name as a reference to the group members is not expressive enough to describe a behaviour that must be triggered by the existence of the facts belonging to different members of the same group. The lack of this expressive power is explained in the remainder of this section and the method of overcoming this problem is explained in the following section.

The following metarule is a fictitious generalised metarule that could appear in the seller-buyers application, if group name were used for generalisation. The generalisation would be achieved by ascribing the metafacts to the group by using the group name (`buyersA`) rather than the names of individual group members.

```
buyersA : 'willing to buy' ::= true &
{ random(80, 100, Price),
  random(100, 200, Amount) }
```

```
==>
buyersA : 'gives $./item to buy . items'(Price, Amount) ::= true
--- start.
```

This metarule would describe determining the product `Amount` to be requested from the seller and the `Price` to be offered for the product. The embedded Prolog call is used to determine those values randomly within a certain range. At the time of the simulation, this metarule is personalised for every member of the `buyersA` group that is in the state of `willing to buy` by replacing all the occurrences of `buyersA` with the actual member name. The following metarule is a personalised instance of this general metarule for the `buyer2` member that belongs to the `buyersA` group.

```
buyer2 : 'willing to buy' ::= true &
{ random(80, 100, Price),
  random(100, 200, Amount) }
==>
buyer2 : 'gives $./item to buy . items'(Price, Amount) ::= true
--- start.
```

Suppose that more than one metafact appearing in the left-hand side of a metarule is ascribed to a group. Suppose too that all these metafacts are expected to belong to the same group member. In such a situation the use of the group name to generalise that metarule would be appropriate. The metarule would describe a behaviour which depends on the state of one group member only. However, there may be other behaviour in the decision problem, which responds to metafacts belonging to different members of the same group. Such behaviour can not be described by metarules that are generalised by the use of the group name. Consider the following partial left-hand side of a fictitious metarule describing such a situation.

```
.... &
buyersA : 'willing to buy' ::= true &
buyersA : 'bought products' ::= true &
....
```

This metarule is expected to be triggered only if one member of the `buyersA` group is in the `willing to buy` state and *another* member is in the `bought products` state. When a metafact ascribed to such a member matches one of the above metafacts, this metarule is personalised by replacing all occurrences of the group name with this member's name. For instance, the existence of the following metafact in the system,

```
buyer2 : 'willing to buy' ::= true
```

will cause the personalisation of the generalised metarule as follows:

```
.... &
buyer2 : 'willing to buy' ::= true &
buyer2 : 'bought products' ::= true &
....
```

Therefore, if the group name is used as a reference to the group members, the personalisation process of a generalised metarule does not allow metafacts in the left-hand side to belong to different members of the same group. In order to enable the description of such behaviour by generalised metarules, there is a need for a more expressive approach, which is described in the following section.

4.1.2.2. References to groups

References to groups denote group members in describing common behaviour. The definition of group references is presented to the system with a new predicate `group_references`, placed in a case customisation file. The form is this:

`group_references(` *group name*, `[` $reference_1$, $reference_2$, ..., $reference_N$ `] ).`

where *group name* is the same as in `group_members` and each $reference_i$ is a name that may be used as a placeholder for the group members in the generalised metarules. There is no restriction on the number of references that can be defined for a group, as long as they all have unique names and as long as the same name does not appear in reference lists for two different groups. The number of necessary references can be determined by considering the metarule in which the most different members need to be referred to.

The following predicates are taken from the seller-buyers application.

```
group_references( buyers1, [ b1, b2 ] ).
group_references( buyers2, [ b3 ] ).
```

These predicates introduce to the Negoplan system the reference names that will be used to refer the members of the `buyers1` and `buyers2` groups in writing generalised metarules. The first predicate introduces the reference name `b1` and `b2` that are used to refer `buyers1` group members and the second predicate introduces the reference names `b3` that is used to refer `buyers2` group members.

With references, the metarule from the previous section, which describes the process of determining the product amount and the price, can be actually written as follows:

```
b1 : 'willing to buy' ::= true &
{ random(80, 100, Price),
  random(100, 200, Amount) }
==>
b1 : 'gives $./item to buy . items'(Price, Amount) ::= true
--- start.
```

The use of reference `b1` instead of the actual group name `buyers1` does not extend the interpretation of this metarule beyond what we have discussed in the previous section. The difference becomes visible at the time of the personalisation of the generalised metarule. All occurrences of the reference in the metarule are replaced by the name of the member, taken from a matching metafact. The other references to the same group are *not* replaced, and they may be further replaced by *other* matching metafacts. Therefore it becomes possible to describe the behaviour that requires the existence of the facts belonging to different members of the same group. The partial left-hand side of the metarule that has been presented in the previous section and that does not describe the intended behaviour can be written as follows:

```
.... &
b1 : 'willing to buy' ::= true &
b2 : 'bought products' ::= true &
....
```

When a metafact ascribed to any of the `buyers1` group members matches one of the generalised metafacts, above, this metarule is personalised by replacing only the occurrences of the reference that appear in the matched metafact; the occurrences of the other references are not affected by this personalisation step. For instance, the existence of the following metafact in the system.

```
buyer2 : 'willing to buy' ::= true
```

will cause a partial personalisation of the generalised metarule by replacing all the occurrences of the reference `b1` with the actual name of the member `buyer2`, as follows:

```
.... &
buyer2 : 'willing to buy' ::= true &
b2 : 'bought products' ::= true &
....
```

This personalisation allows another metafact to be matched by a metafact in the system that is ascribed to any member of the group `buyers1`. As a result of the use of references,

the second metafact can match an existing metafact ascribed to `buyer1` or `buyer2`. Since the original intention of this metarule is for the two metafacts to belong to different members of the `buyers1` group, and since it is still possible to match a metafact ascribed to `buyer2`, additional control is necessary. In order to describe the intended behaviour. an embedded Prolog call can disable the metarule if the two buyers *are* identical:

```
.... &
b1 : 'willing to buy' ::= true &
b2 : 'bought products' ::= true &
{ b1 \== b2 } &
....
```

This is possible because all occurrences of a reference, including the ones in embedded Prolog calls, are replaced with an actual member name.

As a result of the use of references, also in embedded Prolog calls, the expressive power of metarules allows the description of the behaviour that requires the existence of the metafacts belonging to different members of the same group. Therefore, the use of references is more powerful than the use of group name to refer to the group members. That's why this extension has been implemented in the Negoplan system.

### 4.1.3. MODIFICATIONS IN THE USER INTERFACE

The existing Negoplan system on Macintosh computers provides a window for each agent participating in the decision-making problem in order to display their corresponding metafacts. There are three sides, and therefore three windows. The user interface had to be extended to allow windows for all participants.

The extension of the user-interface is based on the predicate `them` that defines the opponents participating in the problem. If the argument of this predicate is a single name that defines the existence of one opponent agent, then the extensions are not activated and the user-interface includes three windows. However, if the argument of this predicate is a list of names that define the existence of more than one opponent, then the extensions are activated and user-interface is modified in order to display one window for each of the opponents. The windows of the opponents are displayed in cascade mode among themselves, because of the screen's space limitation.

A new menu called `Others` is added to the menu bar. It allows the user to hide and show the windows of the selected opponents. This gives the user an ability to make observation on the opponent of the most interest to the user at different times.

In order to give a name to each of the opponent's windows, the meaning of the `side_window` predicate which is used for the opponent is also extended:

`side_window('Other',[`*opponent name*$_1$, ..., *opponent name*$_N$`]).`

where each *opponent name*$_i$ should be replaced with a name representing the opponent defined in the `them` predicate. Therefore, the length of the list in the `them` predicate should be the same as the length of the list in the `side_window` predicate with the first argument `Other`.

The following `side_window` predicate defines the names that are ascribed to the windows of the opponents participating in the seller-buyers application.

```
side_window('Other',[buyer1,buyer2,buyer3,buyer4,buyer5]).
```

The names in the `side_window` predicate do not need to be the same as the names in the `them` predicate for the opponents.

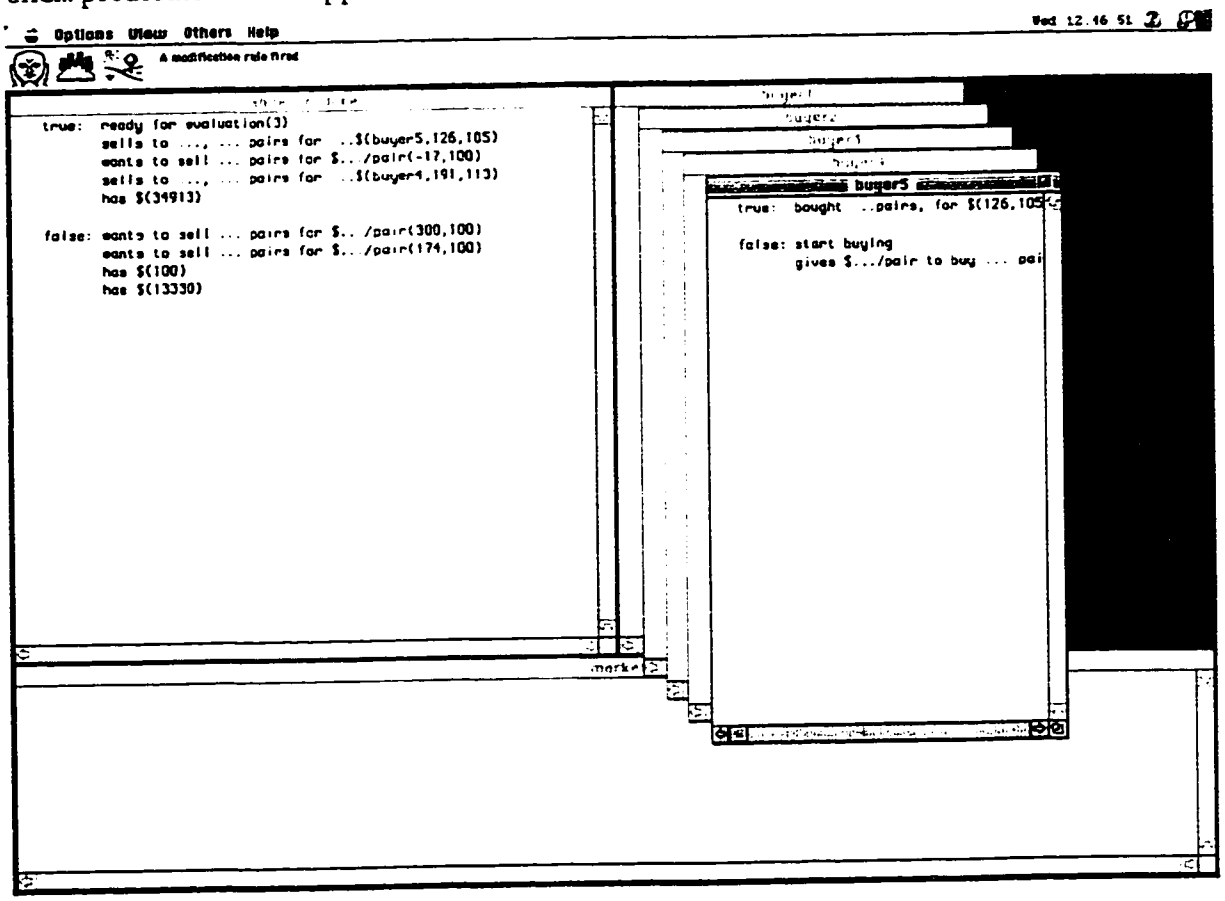



Figure 3. A snapshot from the seller-buyers simulation

Figure 3 is a snapshot from the execution of the seller-buyers application. It depicts the time when the seller sold 126 pairs of shoes to `buyer5` for $105 and 191 pairs of shoes to `buyer4` for $103. At this time, the seller has accumulated $34,913 and is not left with any other product to sell, therefore the application is about to terminate. The buyers, `buyer4` and `buyer5` are not anymore willing to buy the product, because they already bought some. However, although the other buyers, `buyer1`, `buyer2`, and `buyer3` are still willing to buy some products from the seller, they will not be able to, because the products of the seller are sold out.

## 4.2. MODIFICATIONS IN THE INDEXING AND FORWARD CHAINING MECHANISMS

The modifications in the existing constructs and addition of new constructs to the Negoplan system require changes in the indexing mechanism and in the forward chaining mechanism of Negoplan, in order for these modifications to be considered at run time.

### 4.2.1. MODIFICATIONS IN THE INDEXING MECHANISM

Metarules are indexed at metabase compilation time by metafacts that appear in their left-hand sides. When an existing metafact that belongs to an opponent which is a group member is searched in the metarule index, it is not enough to look only for an actual opponent name. Such a metafact can also match the metafacts that appear in the generalised metarules with group references as side names. Therefore, the metafact should also be searched in the index by replacing its opponent's name with every reference that is defined for the group of that opponent. The metarules found after searching the metafact with its actual opponent name and with each reference of its opponent's group constitute the set of metarules that have this metafact in their left-hand sides.

It may be inefficient to search for metarules that have an existing metafact of a group member match metafacts in their left-hand sides, because such a metafact needs to be searched once with its actual opponent's name and once with every reference defined for its opponent's group. In order to avoid this inefficiency, the generalised metarules are indexed for the metafact ascribed to a group after replacing the reference with its group name. This gives one index element for all metafacts ascribed to the same group. As a result of this modification, such a metafact should be searched in the index once with its actual opponent name and once with the group name of its opponent.

For instance, the following metarule index could be created for the following two fictitious metafacts that are ascribed to the buyers1 group members by using different references and that appear in two different response metarules in the same packet:

```
(in metarule 5)      b1 : 'bought products' ::= true
(in metarule 7)      b2 : 'bought products' ::= true
rule_index( buyers1 : 'bought products' ::= true,
                                [ [ packet, [5,7], [] ] ]).
```

An occurrence of the fact bought products with an actual opponent name in the left hand-side of a metarule will result in the creation of a new entry in the metarule index, as follows:

```
(in metarule 9)      buyer1 : 'bought products' ::= true
rule_index( buyer1 : 'bought products' ::= true,
                                [ [ packet, [9], [] ] ]).
```

An alternative to this approach could be to have only one index entry for the same fact that is ascribed to different group members and to the group by using references. In this index entry, the fact could be ascribed to the group by using the group name. With this approach the metarules returned for an existing metafact ascribed to a group member would also include the metarules with the same fact ascribed to different group members, although those metafacts do not match the existing metafact because they belong to different group members. In order to avoid this problem, it is preferred to have one index entry for the same metafacts ascribed to the group and one index entry for each metafact that is ascribed to different group members.

### 4.2.2. Modifications in the forward chaining mechanism

The first major modification implemented in the forward chaining mechanism occurs in searching the metarules that contain a metafact in their left-hand sides matching an existing metafact. The generalised metarules are indexed differently for the metafacts ascribed to the groups and for those ascribed to actual group members. Therefore, an existing metafact that belongs to a group member needs to be searched in the index once to find the metarules that are specifically related to that member and once to find the metarules generalised for that group's members.

The following metafact is examined by the forward chaining mechanism in order to find the metarules that contain a matching metafact in their left-hand sides.

```
buyer1 : 'bought products' ::= true
```

Since `buyers1` is a group member, this metafact will be searched in the index once with its actual member name, `buyer1` and once with its group name, `buyers1`. As a result of the first search the following index entry will be found.

```
rule_index( buyer1 : 'bought products' ::= true,
                                [ [ packet, [9], [] ] ]).
```

As a result of the second search, the following index entry will be found.

```
rule_index( buyers1 : 'bought products' ::= true,
                                [ [ packet, [5,7], [] ] ]).
```

The metarules (metarule 9) that are found by the first search are put on the agenda after simply removing the occurrence of the existing metafact from their left-hand sides. However, the metarules (metarule 5, 7) that are found by the second search are put on the agenda as they are, without removing the matching metafacts from their left-hand sides. This will be done when the metarule is being personalised on the agenda.

The second major modification is for personalising the generalised metarules. Let an existing metafact M ascribed to a group member match a metafact ascribed to that group by a reference in the left-hand side of a generalised metarule and let this metarule not be on the agenda yet. The metarule is put on the agenda without being personalised. If it is already on the agenda because of another existing metafact, it stays on the agenda, without being changed. After all the metarules containing metafacts that match the metafact M have been put on the agenda, the left-hand sides of the metarules on the agenda are parsed. If a metarule has a metafact matching M in its left-hand side and if that metafact is ascribed to the group of M, that metarule is duplicated and its copy is personalised for the member that appears in M. After this duplication process of the generalised metarules, the forward chaining engine removes M from the left-hand sides of the personalised metarules. Therefore after the modifications, on the agenda as well as the personalised metarules, there may also appear some generalised metarules. The existence of the generalised metarules on the agenda gives chance to those metarules to be triggered for the other members of the groups. A generalised metarule that is put on the agenda can be duplicated for a member only once. Therefore, a generalised metarule can be triggered only once for each group member.

### 4.3. Improvement in the Negoplan System Performance

As well as improving the Negoplan system by broadening the decision-making problem domain, the modifications also had some negative impact on the system performance.

The main reason is a significant increase in the number of metafacts existing in the system at a given time. This increase is a result of having more than one opponent in the new decision-making problems. In order to overcome the inefficiency problem that is expected to occur in the application of the new type of problems, three measures have been taken. These are explained in the following subsections.

### 4.3.1. Improvement in the efficiency of the inference engine

In the inference engine, the metarules that contain the current existing metafact in their left-hand sides are determined by using the metarule index. If those metarules have not been already placed on the agenda by another existing metafact, they are placed on the agenda now. The efficiency of this process did not change much because of the approach used in the modification of the indexing mechanism. The number of times the metarule index is searched for an existing metafact has increased only by one and only for the existing metafacts that are ascribed to the group members.

The metarule index was not used, however, to look up the existing metafacts in the metarules already on the agenda. Instead, metafacts in the left-hand sides of the metarules on the agenda were traversed sequentially. Because of duplication, the number of metarules that may appear on the agenda for problems with more than one opponent is expected to be much higher than the number of metarules for problems with one opponent. It has been found more efficient (and implemented) to improve indexing to require traversing only the left-hand sides of the metarules that are known to contain a matching metafact.

### 4.3.2. Modifications of the metafacts

In decision-making problems with more than one opponent, the states of all opponents need to be described by metafacts ascribed to each opponent. Therefore, the number of metafacts is expected to be significantly higher than for problems with one opponent. Apart from this negative impact caused by the extension of the Negoplan system, two metafacts which are exactly the same except for the values of their arguments are also asserted as new metafacts to the system by the current Negoplan implementation.

The following metafacts describe the non-existence of `buyer1`'s willingness to buy 150 items for the price of $99 and for the price of $105 and they may appear in the system at the same time.

```
buyer1:'gives $./item to buy . items'(99,150)::=false
buyer1:'gives $./item to buy . items'(105,150)::=false
```

The existence of the same metafact with different argument values may be informative to the user for some decision-making problem simulation that does not have efficiency concerns, but for some complicated simulations which are expected to have a substantial number of metafacts, it may be necessary to decrease this number as much as possible in order to increase the efficiency of the forward chaining mechanism. This can be partially achieved by preventing the assertion of the same metafacts with different arguments as different metafacts. Instead, a metafact can overwrite another metafact already in the system if the arguments of the old metafact are not important for the rest of the simulation. This will prevent the cumulative increase of the number of metafacts that appear in the system.

The `metafacts_replaced` predicate, introduced in the customisation file, is used to determine the metafacts that will be replaced. If a new metafact matches a partially instantiated metafact pattern introduced by one of the `metafacts_replaced` clauses, the existing metafact is removed and the new metafact with different argument values is asserted.

The following `metafacts_replaced` predicate taken from the seller-buyers application, introduces a metafact that will be replaced every time the same metafact is newly asserted with different argument values.

```
metafacts_replaced(
buyer1 : 'gives $./item to buy . items'(_,_) ::= true).
```

As a result of this definition, if the following metafact exists in the system,

```
buyer1:'gives $./item to buy . items'(99,150)::=false
```

and the following metafact is being newly asserted to the system,

```
buyer1:'gives $./item to buy . items'(105,150)::=false
```

then the existing metafact is first removed and the new metafact is asserted. That is, one copy of this metafact (with different argument values) is present at any time of the simulation. The accumulation of similar metafacts with different argument values is expected to be reduced substantially.

### 4.3.3. MODIFICATIONS IN THE INDEXING MECHANISM

At the beginning of the simulation, the indexing mechanism parses all metafacts on the left-hand sides of all the metarules. When a metafact is examined, first of all it is searched in the index. If there is an entry which contains a metafact that matches the examined metafact, then the metarule number of the examined metafact is appended to that index entry. If there is no such entry, a new entry is created for that metafact. This mechanism works without any problem for the metafacts that do not contain any uninstantiated variables. However, if some metarules contain in their left-hand sides, metafacts that have uninstantiated variables, this mechanism may result in a situation affecting the system performance negatively.

The following two metafacts that appear in the left-hand sides of two different metarules are assumed to be examined in the order they are introduced here.

(in metarule 5) `buyer1:'bought .items, for $'(Amount,Price) ::= true`

(in metarule 7) `buyer1:'bought .items, for $'(143,105) ::= true`

Since the first metafact is the first to be examined, it is inserted as a new index entry. The matching of the metafacts was achieved by using the predefined = operator. At the time of the second metafact's examination, its metarule number is appended to the list of metarules appearing in the index entry that was created for the first metafact, because the index entry that is created for the first metafact contains two uninstantiated variables and matches the second metafact according to the = operator used for matching metafacts. As a result of indexing those metarules, the index entry would be as follows:

```
rule_index(buyer1:'bought .items, for $'(Amount,Price)::=true,
                                        [ [ packet, [5,7], [] ] ]).
```

This causes no problem if there is no other metafact matching the first metafact,

```
buyer1:'bought .items, for $'(Amount,Price)::=true
```

However, if the following metafact also exists in the left hand side of another metarule,

(in metarule 9) `buyer1:'bought .items, for $'(137,110) ::= true`

this metarule will also be indexed by the same index entry because this metafact also matches the metafact of the above index entry, and the index entry will become:

```
rule_index(buyer1:'bought .items, for $'(Amount,Price)::=true,
                                        [ [ packet, [5,7,9], [] ] ]).
```

With this index entry, at the time of the forward chaining mechanism as a result of the following existing metafact,

```
buyer1:'bought .items, for $'(143,105) ::= true
```

the metarules 5, 7 and (unnecessarily) 9 will be put on the agenda. Having an extra metarule in the agenda is not an error, because that extra metarule is not triggered unless there are really existing metafacts that match the metafacts in its left-hand side. However, when the decision-making problem is complicated, this may cause negative impact in the performance of the system.

In order to avoid this problem, the matching process of the metafacts is achieved by a user-defined === equality operator, a variant of the built-in == that matches different variables. As a result of this operator, a new index entry is created for the examined metafact unless there is already an index entry exactly for that metafact. With the use of this operator, as a result of the three metafacts that appear in metarules 5, 7 and 9, three metarule index entries are created, as follows:

```
rule_index(buyer1:'bought .items, for $'(Amount,Price)::=true,
                                        [ [ packet, [5], [] ] ]).
rule_index(buyer1:'bought .items, for $'(143,105)::=true,
                                        [ [ packet, [7], [] ] ]).
rule_index(buyer1:'bought .items, for $'(137,110)::=true,
                                        [ [ packet, [9], [] ] ]).
```

Now, in order to find the metarules that depend on an existing metafact M, the index should be accessed once for every entry that unifies with M. The metarule numbers collected as a result of all accesses determine the set of metarules that depend on M.

If the existing metafact is,

```
buyer1:'bought .items, for $'(143,105) ::= true
```

the metarule numbers that are listed in the first two metarule index entries are collected and as a result the metarules 5 and 7 are put on the agenda.

As a result of this approach, addition of metarules on the agenda that are not dependent on the existing metafact is prevented and the process of parsing the metarules that are on the agenda is expected to be more efficient.

## 4.4. TESTING THE EXTENSIONS

The extended Negoplan system has been tested for two main reasons. The first reason is to find out if the execution of the existing decision-making problem applications that do not use the extensions is affected by the implementation of the extensions. This has been achieved by executing the earlier decision-making problem applications of the Negoplan system without the extensions and with the extensions. The behaviour of the simulations that are obtained as a result of executing the same problem on different Negoplan systems have been compared. When all the earlier simulations are observed to behave exactly the same way on the two Negoplan systems with the extensions and without the extensions, it is concluded that the extensions do not interfere with any of the existing constructs of the Negoplan system.

The second reason is to find out if the implemented extensions provide the behaviour that is expected of the system. In order to determine this, an application for a small decision-making problem involving more than one opponent has been written in the extended Negoplan and the simulation of this problem has been observed to see whether *all* the extensions have been successfully implemented.

The test application involves a company (`seller`) that wants to sell its products and five other companies (`buyer1`, `buyer2`, `buyer3`, `buyer4` and `buyer5`) that are continuously making offers until buying the required amount of those products. All of the offers are presented to the `seller` company at the same time. The buyers whose offers are accepted by the `seller` company buy the amount of products they requested and the buyers whose offers are not accepted need to prepare a new offer for the next evaluation of the `seller` by increasing the price they are offering in order to increase the chance of their offers being accepted. The simulation continues until either all the companies have bought the required amount of products or all the products have been sold out.

There are two groups of opponents. The first group includes `buyer1` and the `buyer2`. The second group includes `buyer3`, `buyer4` and `buyer5`. These groups are introduced to the Negoplan system with the following predicates:

```
group_members( buyers1, [ buyer1, buyer2 ] ).
group_members( buyers2, [ buyer3, buyer4, buyer5 ] ).
```

The following references are defined in order to refer the members of each group in the metarules describing the common behaviour of the group members.

```
group_references( buyers1, [ b1, b2 ] ).
group_references( buyers2, [ b3 ] ).
```

The behaviour of the different group members varies in the determination of the price that will be offered for the products. The prices that are offered are determined considering different ranges and if those prices are not accepted by the seller, the increases again are determined differently.

The problem is viewed from the `seller`'s point of view and the initial problem decomposition graph introduces a `seller` who has initially `$100` and who wants to sell `300` items of its products for at least `$100` per item.

```
seller <-   'has $'(100) &
            'sells . items for $./item'(300,100).
```

The simulation starts with all the opponents willing to buy products from the `seller`. The amount of products they need and the price that will be offered for one item are determined by the following generalised metarules for each group.

```
b1 : 'willing to buy' ::= true &
{ random(60,100, Price),
  random(100,200, Amount) }
==>
b1 : 'gives $./item to buy . items'(Price, Amount) ::= true
--- start.

b3 : 'willing to buy' ::= true &
{ random(80,100, Price),
  random(100,200, Amount) }
==>
b3 : 'gives $./item to buy . items'(Price, Amount) ::= true
--- start.
```

The first metarule describes the behaviour of the `buyers1` group and the second metarule describes the behaviour of the `buyers2` group. The only difference between their behaviour is the range from which they determine the initial price that will be offered to the `seller`. Since the members of the `buyers2` group determine their price offer within the range of `80` and `100`, acceptance of their offers is more likely than the acceptance of the offers of the `buyers2` members.

After having determined the price and the amount that will be requested, the simulation goes into an evaluation phase in which the `seller` accepts or rejects the offers after considering its price criteria.

```
b1:'gives $./item to buy . items'(B_Price,B_Amount)::=true &
{ existing_metafact( seller,
            'sells . items for $./item'(S_Amount,S_Price), true),
  S_Amount > 0,
  B_Price > S_Price,
  New_S_Amount is S_Amount - B_Amount }
==>
seller:'sells . items for $./item'(New_S_Amount,S_Price)::= true &
seller:'sells . items for $./item'(S_Amount,S_Price) ::= false &
seller : 'sells to ., . items for .$'(b1,B_Amount,B_Price) ::= true &
b1 : 'gives $./item to buy . items'(B_Price,B_Amount) ::= false
--- evaluation.
```

The above metarule describes the conditions that need to be satisfied in order to have a deal between the `seller` and a buyer and also the changes that will appear in the states of the two parties as a result of this interaction. This metarule only describes the interaction between the `buyers1` group members and the `seller`. The same interaction between the `buyers2` group members and the `seller` is described by another metarule in which only the references (`b1`) to the `buyers1` group members are replaced with a reference to the `buyers2` group members.

This metarule can be triggered once for every `buyers1` group member with the conditions that the `seller` still has some more products to sell and the offered price is greater than the price expected by the `seller`. As a result of this metarule, the new amount of products that needs to be sold by the `seller` is updated, the buyer is not anymore interested in buying more products and a new metafact ascribed to the `seller` is added to the existing metafacts describing the deal that occurred between the `seller` and the buyer. For instance, if the `buyer2`'s offer, which is `$120`/item to buy `135` items, is accepted by the `seller`, since at the time of the personalisation of this metarule all the references (`b1`) are replaced with `buyer2`, the following metafact will be added to the existing metafacts.

```
seller : 'sells to ., . items for .$'(buyer2,135,120) ::= true
```

The following metarule describes how the money owned by the `seller` is updated after a deal has been achieved between the `seller` and any `buyers1` group member.

```
b1 : 'gives $./item to buy . items'(B_Price,B_Amount) ::= true &
seller : 'sells to ., . items for .$'(b1, B_Amount, B_Price) ::= true &
```

```
{ existing_metafact( seller, 'has $'(S_Money), true),
  New_S_Money is S_Money + B_Price * B_Amount }
==>
seller : 'has $'(New_S_Money) ::= true &
seller : 'has $'(S_Money) ::= false
--- evaluation.
```

After all the offers have been considered by the `seller`, the simulation either continues with the next phase in which the buyers increase their offers or terminates as a result of nothing being left for sale or no more products requested by the buyers.

The termination of the simulation is defined by the following metarules.

```
seller : 'wants to sell . items for $./item'(S_Amount,S_Price)::=true &
{ S_Amount < 0 }
==>
terminate 'Everything is sold'
--- evaluation.
```

```
seller : 'wants to sell . items for $./item'(S_Amount,S_Price)::=true &
{ \+ existing_metafact(
      Buyer,'gives $./item to buy . items'(B_Price,B_Amount), true) }
==>
terminate 'No more offers'
--- evaluation.
```

The first termination metarule is triggered when the amount that needs to be sold becomes a negative value. The second termination metarule is triggered if there is no more buyer willing to buy products from the `seller`.

If the `seller` has more products to sell and there are more buyers who are willing to buy in the system, then the simulation goes to the phase in which the buyers reconsider their offers, with the following restructuring metarule. `'ready for evaluation'(N)` fact is used for the purpose of counting the total number of evaluations that will appear until the end of the simulation.

```
seller : 'sells . items for $./item'(S_Amount,S_Price)::=true &
seller : 'has $'(S_Money) ::= true &
seller : 'ready for evaluation'(N) ::= true &
{ existing_metafact(Any_Buyer,
      'gives $./item to buy . items'(B_Price,B_Amount), true),
```

```
  S_Amount > 0 }
==>
modify (
seller <-   'sells . items for $./item'(S_Amount,S_Price) &
            'has $'(S_Money) &
            'ready for evaluation'(N) )
switch_to buyers_are_bidding
--- evaluation.
```

The following metarules describe how the price offered per item is increased by the members of the two groups. According to these metarules, the price is increased by the `buyers1` group members in the range of `1` and `10` and by the `buyers2` group members in the range of `1` and `20`. Since the `buyers2` group members increase their price offers more than the `buyers1` group members, it is more likely that their offers will be accepted by the `seller`.

```
b1 : 'gives $./item to buy . items'(Price,Amount) ::= true &
{ random(1,10, Increase),
  New_Price is Price + Increase }
==>
b1 : 'gives $./item to buy . items'(New_Price,Amount) ::= true
--- buyers_are_bidding.

b3 : 'gives $./item to buy . items'(Price,Amount) ::= true &
{ random(1,20, Increase),
  New_Price is Price + Increase }
==>
b3 : 'gives $./ item to buy . items'(New_Price,Amount) ::= true
--- buyers_are_bidding.
```

As a result of the simulation of this decision-making problem, the interaction between the `seller` and the buyers appeared as expected and the new Negoplan system has been found ready for more complicated problems.

# 5. THE BUSINESS STRATEGY GAME APPLICATION

The Negoplan system is used for decision support and allows the definition of autonomous agents by describing their behaviour in detail. Negoplan can be customised for the managerial decision-making problem addressed by the BSG software, and provide an environment in which the user can practice his theoretical knowledge. Unlike the BSG software, Negoplan can create such an environment without the availability of other teams. The user of the managerial decision-making problem application can be represented by the supported agent of the system and can compete against the other management teams of the companies that are simulated by opposing agents.

Section 5.1 presents the complexities of the BSG software which are not essential for achieving the purpose of the BSG application. Section 5.2 gives a brief introduction to the BSG application and Section 5.3 introduces the flow of the model. Some technical details of the application is explained in Section 5.4. The strategies that can be followed by all the teams are described in detail in Section 5.5 and the methods used in the evaluation of these strategies are explained in Section 5.6. The problem decomposition graph (PDG) is introduced in Section 5.7. The decision-making processes of the user and the opposing management teams are presented in the Sections 5.8 and 5.9 and the changes that occur in the environment during the simulation are presented in Section 5.10. This chapter is concluded with a review in Section 5.11.

## 5.1. VARIETIES OF THE BUSINESS STRATEGY GAME

The variations in the BSG software allow more than one manufacturing plant and more than one market with growing demand. These variations make the game very realistic and practically very educational from its user's point of view. Depending on the state of the environment and the other companies, a decision that might be very beneficial in one situation may result in a loss in some other situations. For instance, it could be advantageous to sell the product in a market in which the tax rate is lower than the others, or it could be disadvantageous to ship high quality products to a geographic market where high quality products are not in high demand. As a result of all this variety, the management teams are not only concerned with one set of decisions. They need to make decisions for every plant they own and for every market in which they sell their products. The teams also need to decide which products of which plant will be shipped to which geographical market in order to maximise benefits. It is possible for more than one management team to be successful at the same time and to have different geographic

markets. These are some of the consequences of the context-dependent property of the managerial decision-making problems.

Theoretically, Negoplan constructs are expressive enough for the representation of the behaviour of any number of competing management teams. Therefore, technically there is no obstacle to the BSG application. However, the variety, while making the game very realistic, results in so many alternatives that the implementation of the application becomes practically almost impossible with the present performance of the Negoplan system.

Besides implementing a training tool, another purpose of the BSG application is to show the use of the extensions and the usefulness of the Negoplan in managerial studies. Therefore, putting all aspect of the BSG into the simulation would not add more value to the application. As a result, some aspects of the BSG software have been simplified, in order to shorten the time required to achieve the purposes of the application.

In order to simplify the game, the number of plants a team can own and the number of geographical markets that exist in the game are reduced to just one plant in the same geographical market which still has growing demand for products. As a result of the simplification, the number of decisions a management team needs to make has been reduced substantially, as well as the alternatives that need to be considered in making those decisions. The decisions presented in Section 2.1 still need to be made but only for just one plant.

## 5.2. The Business Strategy Game Application of Negoplan

The Negoplan system has been customised to represent the management teams of several athletic footwear manufacturing companies that make annual decisions in order to become the leader of the industry. This decision-making problem is viewed from one team's point of view and that team is supported by the Negoplan system. The state of the company managed by that team is described by a Problem Decomposition Graph (PDG)—see Section 5.7. The other management teams are represented by the opposing agents that behave autonomously. Since their objectives and the decisions that they need to make are the same, the other management teams are assumed to behave essentially in exactly the same way. Therefore, they are grouped together so that their behaviour can be described by generalised metarules. Some of these metarules are randomised in order to prevent the sameness in the behaviour of these agents and to obtain variations in different executions of the application.

The purpose of the managerial decision-making problem simulation is to provide an environment to the Business Administration students in order to let them exercise their theoretical knowledge. In order to achieve this, the control of the supported agent's decision-making process is fully given to a user (expected to be a student) who would like to test different strategies. After every modification in the student's decisions, some numerical data that are related to the category of that decision are displayed in a window with the decisions of that category, so that the student can make observation on the impacts of these modifications and make some adjustments, if necessary. In the rest of this chapter, "agent" will refer to any team participating in the problem, "student" to the team represented by the supported agent and "opposing agents" to the other teams.

## 5.3. THE FLOW OF THE SIMULATION

There are three main phases in the BSG application of Negoplan (Figure 4). The execution starts with each agent's decision-making process for the following year. These decisions are made without the agents being aware of each other's decisions, as in real life. After all the agents have completed their decisions, in the evaluation phase, the market share percentages of each agent are determined according to the strategies followed. After evaluation, the resetting phase resets the environmental factors considering the results of the previous year. After all these phases have been completed, the execution continues with the decision-making process of the agents for the following year by considering the results obtained in the current year. Therefore, these three phases are repeated until the student selects the `end_game` option as a result of the following selection metarule.

```
student : makes_decision_year_round(_,_) ::= true
==>
select one ( make_high_level_decisions,
             make_low_level_decisions,
             end_decision_making_process,
             end_game )
with_message 'What is your choice ?'
--- student_decision_menu.
```

The selections `make_high_level_decisions` and `make_low_level_decisions` offered in the above metarule are explained in the section 5.9.

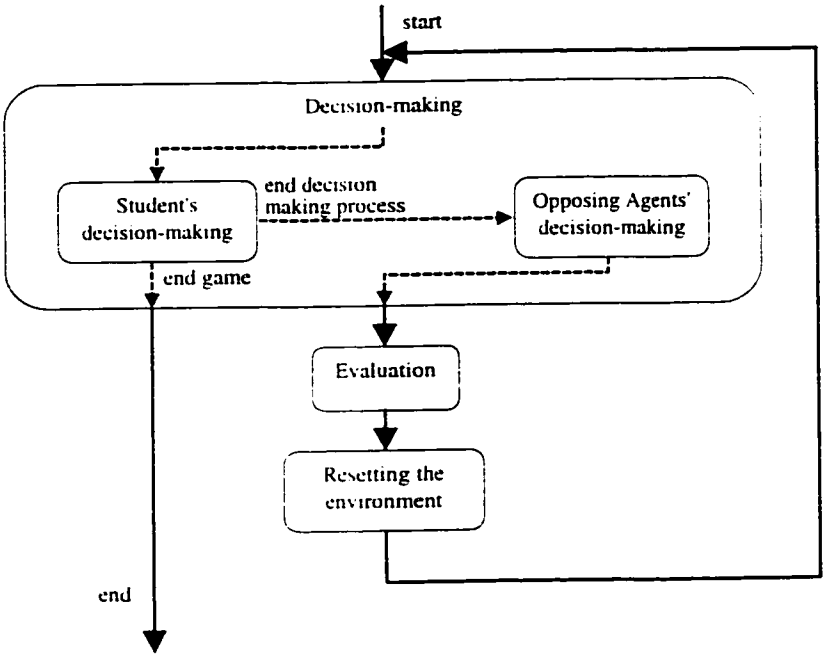



Figure 4. Phases of the Business Strategy Game application

In real life, the decision-making processes for all the agents would possibly occur in parallel. However, because Negoplan is run sequentially, the decision-making processes of the student and the opposing agents need to be done in two sub-phases. First the student makes the decisions and then the opposing agents make their decisions by the activation of the generalised metarules. Since the decisions made by an agent are not observable by the other agents, the parallelism that occurs in real life is preserved. In Figure 4, the flow of phases that occurs in real life is shown using solid lines and the flow of sub- phases that occurs in the simulated decision-making phase is shown using dashed lines.

### 5.4. REPRESENTATION OF THE DECISIONS

In order to calculate the financial indicators and the intermediate data related to different categories of decisions, the necessary operational and financial formulas are included in the application as user-defined Prolog procedures. Those procedures (Appendix C) are invoked in metarules whenever necessary, using the mechanism of embedded Prolog calls.

The quantitative decisions of the agents are used in the calculation of the financial indicators. An agent's decisions can be considered a part of its state, but the decisions of opposing agents can not be stored within metafacts because the metafacts of each agent

are displayed in that agent's window. This is not realistic. The quantitative decisions must be stored using a representation that can not be observed by the user. This is achieved by using the `stored_param2` clause as follows:

`stored_param2(` *agent*, *decision_category*, *list_of_decisions* `).`

The placeholders *agent*, *decision_category* and *list_of_decisions* represent the actual name of an agent, the category for which the decisions are stored and the list of decisions that belong to the specified category, as described in Section 2.1.

A change in the values of a decision category for an agent is made in the `stored_param2` clause corresponding to that decision category and to that agent using the `write_param2` predicate. This predicate simply overwrites an existing matching clause with the new quantitative decisions. Whenever a quantitative decision is necessary to calculate a financial indicator of a certain agent, that decision is retrieved from the appropriate `stored_param2` clause using the `read_param2` predicate.

In the `stored_param2` clauses of each agent that corresponds to the production decisions, labour decisions and marketing decisions, the external factors related to those categories that play a role in the calculation of the financial indicators are stored as the last elements of the *list_of_decisions*. The external factors are discussed in more detail in Section 5.6.

The following `stored_param2` clause is used to store the `student`'s production decisions. The values in the *list_of_decisions* are respectively: pairs to be manufactured, percent use of long-wear material, the number of models, the quality control budget, the styling/features budget, the methods improvement budget and the reject rate external factor.

```
stored_param2( student, production_decisions,
                                [1989, 25, 100, 785, 500, 0, 5.00] ).
```

The values stored in the *list_of_decisions* can be retrieved by following `read_param2` predicate call and assigned to the variables `Quantity`, `Long_wear`, `Models`, `Quality_budget`, `Styling_budget`, `Improvement_budget`, `Reject_rate`, respectively.

```
read_param2(student, production_decisions,
                    [ Quantity, Long_wear, Models, Quality_budget,
                    Styling_budget, Improvement_budget, Reject_rate ] )
```

The decision on pairs to be manufactured can be replaced with the value of the `New_Quantity` variable using the following `write_param2` predicate call.

```
read_param2(student, production_decisions,
                    [ New_Quantity, Long_wear, Models, Quality_budget,
                    Styling_budget, Improvement_budget, Reject_rate ] )
```

The method of evaluating the decisions of the agents, presented in Section 5.6, is based on the average decisions of all the agents. The average decisions are also stored using `stored_param2` clause by replacing the *agent* placeholder with `average`. The following is an example of average labour decisions.

```
stored_param2( average, labor_decisions, [663, 12.0, 1.5, 3000] ).
```

In the rest of this chapter, in the presentation of the metarules, the calls to retrieve and to update the quantitative decisions are discarded for simplicity. It is only shown how those decisions are updated.

## 5.5. STRATEGIES FOR THE AGENTS

The actual BSG highly depends on quantitative decisions, which is not likely to happen in reality. A management team usually makes qualitative decisions which are later converted into some quantitative values by other individuals. In order to provide a realistic decision-making environment, the quantitative decisions that have common impact on the operation of the company are categorised. This helps come up with qualitative decisions that enable the opposing agents to make their decisions autonomously. As a result of categorising the decisions, the management teams can make qualitative decisions rather than quantitative.

The metarules that describe the conversion of a selected qualitative decision for a decision category into its corresponding quantitative decisions are the same for the student and for the opposing agents. In order to avoid presenting these metarules twice in the following subsections, for the student and for the opposing agents, the facts that belong to the agents in those metarules will be ascribed to a generalised side labelled `agent`.

The quantitative decisions of two agents that follow the same strategy in a category are assumed to be modified exactly the same way. However, identical strategy conversion metarules may result in identical decisions for the agents with the same strategies. In order to avoid having agents with exactly the same decisions, modifications are done randomly within certain ranges, resulting in some variation.

The first criterion for categorising the decisions is the time when they become effective. Some of the decisions, like the number of pairs of shoes to be manufactured, become

effective in the year immediately following the decision. Other decisions, like the automation option, become effective the year after they have been made. As a result of this distinction, the agents are supposed to make their decisions considering their short-term and long-term plans.

Although the financial decisions become effective next year, they will not be considered as short-term plans, because the financial decisions are not directly related to the decisions that change the image of the company in the market. The financial decisions are made by considering the expenses and the savings that occur as a result of other decisions.

The short-term qualitative decisions, long-term and financial decisions represent the strategy of an agent.

### 5.5.1. SHORT-TERM QUALITATIVE DECISIONS

All the decisions, other than plant decisions and financial decisions become effective in the year immediately following, therefore they reflect the short-term plans of the agents. However, those decisions do not all address the same aspect of managing the athletic footwear manufacturing company. While some decisions are related to manufacturing, others are related to the marketing. Because of this, classifying all these decisions only as short-term decisions would not enable the agents to make meaningful decisions. Therefore, there is a need to further categorise the short-term decisions.

As a result of investigating the consequences of each decision, four different categories are identified in the short-term decisions. These are quality, service, advertisement and price. As a result of this subcategorisation, the agents' behaviour can be different in those decision subcategories.

This behaviour of the agents is defined as qualitative decisions, as follows:

Three strategies can be followed for these new categories.

Conservative : When the behaviour is conservative in one of the categories, the agent is understood not to want to make investment to be the industry leader in that category. This situation may occur when there are not enough financial resources to invest in more than one category. As a result of being conservative in a category, the agents may expect to be positioned as an average company in that category, however if other agents are less conservative in the same category, being conservative may lead to a situation that is worse than average.

Aggressive : When the behaviour is aggressive in one of the categories, the agent is understood to want to become the industry leader in that category. Since the objective of every agent in the market is to become a leader, the agents are expected to be aggressive in all the categories. However, since being aggressive requires making investment for a category, the agents may not have enough financial resources to be aggressive in all the categories. Therefore, the agents are usually expected to select the categories for which they want to be aggressive. By being aggressive, the agents may expect to be positioned as a leading company in the selected categories, however if the other agents are also aggressive in the same category, being aggressive may lead to a situation that is worse than being a leader.

Reactive : Being reactive is interpreted as the agent's attempt to improve its position which is lower than average in some category. An agent which is lower than average in a category is considered to be in a critical situation, therefore the agent needs to take measure in order to improve its situation. This is done by investing more than when being aggressive in order to catch up to the competing agents. As a result of being reactive in a category, the agents may expect to be positioned as an average company in that category, however if other agents are conservative, being reactive may even give a leading position in that category.

### 5.5.2. FROM QUALITATIVE DECISIONS TO QUANTITATIVE DECISIONS

The qualitative decisions for the four categories are converted into quantitative decisions based on the average quantitative decisions of all the agents, made in the previous year. When an agent is conservative in a category, the quantitative decisions that correspond to that decision will be the average quantitative decisions of the previous year for that category. When an agent is aggressive or reactive, the quantitative decisions will be the average multiplied by a certain coefficient. Those coefficients are higher when being reactive because the agent's objective is to improve its state (it is worse than the state of an average agent).

The qualitative decisions of the agents are stored in metafacts of the following form, where *agent* is replaced by an actual agent name, and *price*, *quality*, *service* and *advertisement* are replaced by specific qualitative decisions (conservative, aggressive or reactive).

*agent* : `short_term_strategy(` *price*, *quality*, *service*, *advertisement* `) ::= true`

In the metarules that describe the conversion of qualitative decisions into quantitative decisions, the average values that are likely to be real are converted into values having a certain number of digits after decimal point by using the pre-defined `integer` predicate. The predicate `strategy_coefficient` is used to define coefficients for different categories and for their defined qualitative decisions that will be used in converting the qualitative decisions into their corresponding quantitative decisions. The quantitative decisions of a qualitative decision for a category is calculated by multiplying the previous year's average decisions by the coefficient corresponding to that qualitative decision and category. The definitions of these coefficients can be updated by simply changing the values defined by `strategy_coefficient` predicate. The coefficients that are used in the rest of this section are selected so that they result in decisions within reasonable ranges. The following is an example definition which introduces `1.05` as the coefficient for the `quality` category and an aggressive qualitative decision.

```
strategy_coefficient(quality, aggressive, 1.05).
```

The metarules describing the conversions of the short-term qualitative decisions are triggered immediately after they are made.

In the presentation of the metarules, only the essential parts related to the conversion of a qualitative decision to its corresponding quantitative decisions will be presented. The parts that access and make the changes in the quantitative decisions of the agents can be found in the full text of the metarules (Appendix B).

#### 5.5.2.1. Qualitative price category decision conversions

Decisions on the methods improvement budget, the wholesale price to dealers and the customer rebates are associated with the price category. The wholesale price decision is made after all the decisions because it depends on the decisions that affect the cost of production. This decision is made separately, after all the decisions are made. The wholesale price is determined by multiplying the operational cost per pair produced by the coefficient corresponding to the qualitative category decision made. Since when being aggressive and reactive the sale amount is expected to be higher than when being conservative, the wholesale price should be lower than when being conservative. This is satisfied by associating the conservative decision with a higher coefficient than the aggressive and reactive decisions. The coefficients for conservative, aggressive and reactive decisions are 1.3, 1.27 and 1.24, respectively.

When the rebate amount is higher, the sale amount is expected to be higher, therefore the aggressive and the reactive decisions result in greater rebate amounts. The predicate `new_rebates_wrt_strategy` returns the new rebate amount that will be returned to the customers. Since the rebate amount can be 0, 1, 3 or 5, this predicate returns a value greater than the average rebate amount when the qualitative decision is aggressive or reactive and a smaller value when the qualitative decision is conservative.

The methods improvement budget is an investment decision that has an increasing effect on the operational cost, but it is considered as a part of the price category, because it also has a reducing effect on the fixed costs of production. This decision is made by multiplying the methods of improvement budget by the corresponding coefficient of the qualitative price category decision made. When the qualitative decision is aggressive and reactive, the amount invested for methods improvement is less than when it is conservative in order to keep the operational cost as low as possible, so that the wholesale price is lower.

The following partial metarule describes the conversion of the qualitative price category decision into its corresponding quantitative decisions, except for the wholesale price decision.

```
agent : price_strategy_updated ::= true
==>
{ ...,
  new_rebates_wrt_strategy(Avg_Rebates, Price_Strategy, New_Rebates),
  strategy_coefficient( price, Price_Strategy, Coeff ),
  New_Improvement_budget is integer(Avg_Improvement_budget * Coeff),
  ...
} &
agent : price_strategy_converted ::= true
--- agent_makes_high_level_decisions.
```

5.5.2.2. Qualitative quality category decision conversions

Decision on the percentage of long-wear material, the number of models, the quality control budget, the styling/features budget, the annual wages and the incentive pay per pair are associated with the quality category. The following partial metarule describes the conversion of a qualitative quality category decision into its corresponding quantitative decisions. The coefficients for conservative, aggressive and reactive decisions are 1, 1.05 and 1.1, respectively. The predicate `new_models_wrt_strategy` returns the number of

models that will be produced according to the qualitative decision made and the average number of models produced by all agents. The number of models can be 50, 100 or 200; this predicate returns a value greater than the average number of models if the qualitative decision is conservative, and a value smaller than the average for the other two qualitative decisions.

```
student : quality_strategy_updated ::= true
==>
{ ...,
  strategy_coefficient( quality, Quality_Strategy, Coeff ),
  New_long_wear is integer(Avg_long_wear * Coeff),
  new_models_wrt_strategy( Avg_Models, Quality_Strategy, New_models ),
  New_quality_budget is integer(Avg_quality_budget * Coeff),
  New_styling_budget is integer(Avg_styling_budget * Coeff),
  ...,
  New_annual_wage is integer(Avg_annual_wage * Coeff * 10) / 10,
  New_incentive_pay is integer(Avg_incentive_pay * Coeff * 10) / 10,
  ... } &
agent : quality_strategy_converted ::= true
--- agent_makes_high_level_decisions.
```

5.5.2.3. Qualitative service category decision conversions

Decisions on the number of service representatives, the number of retail dealers and the delivery time are associated with the service category. The following partial metarule describes the conversion of a qualitative service category decision into its corresponding quantitative decisions. The coefficients for conservative, aggressive and reactive decisions are 1, 1.05 and 1.1, respectively. The predicate `new_delivery_wrt_strategy` returns the time period required to deliver products to the dealers according to the qualitative decision made and the average delivery time of all the agents. Since the delivery time can be 1, 2, 3 or 4, this predicate returns a value smaller than the average delivery time when the qualitative decision is aggressive or reactive and a greater value when it is conservative.

The number of dealers is decided by simply multiplying the average number of dealers by the coefficient corresponding to the qualitative service category decision made. However, since the impact of the number of service dealers depends on the dealers per service representative ratio, first of all this ratio is calculated. Since it should be decreased

in order to have an improvement in the service, it is multiplied by the inverse coefficient. For instance, if the ratio is 1.05, the inverse coefficient is approximately 0.95, as a result of dividing 1 by 1.05.

```
agent : service_strategy_updated ::= true
==>
{ ...,
  strategy_coefficient( service, Service_Strategy, Coeff ),
  Inv_Coeff is 1/Coeff,
  New_Dealers is integer(Avg_Dealers * Coeff),
  Avg_Dealers_per_serv_rep is integer(Avg_Dealers / Avg_Service_rep),
  New_Dealers_per_serv_rep is Avg_Dealers_per_serv_rep * Inv_Coeff,
  New_Service_rep is integer(New_Dealers / New_Dealers_per_serv_rep ),
  new_delivery_wrt_strategy( Avg_Delivery ,Service_Strategy,
                                                    New_Delivery ),
  ... } &
agent : service_strategy_converted ::= true
--- agent_makes_high_level_decisions.
```

5.5.2.4. Qualitative advertisement category decision conversions

Only the advertising budget decision is associated with the advertisement category. The following partial metarule describes the conversion of the qualitative advertisement category decision into its corresponding quantitative decisions. The coefficients for conservative, aggressive and reactive decisions are 1, 1.25 and 1.5, respectively. These coefficients are higher than for other categories, in order to obtain a noticeable difference as a result of the change in the qualitative decision. The new budget for advertisement is calculated by simply multiplying the average advertisement budget of the previous year by the coefficient corresponding to the qualitative advertisement category decision made.

```
agent : advertisement_strategy_updated ::= true
==>
{ ...,
  strategy_coefficient( advertisement, Adv_Strategy, Coeff ),
  New_Adv_budget is integer(Avg_Adv_budget * Coeff ),
  ... } &
agent : advertisement_strategy_converted ::= true
--- agent_makes_high_level_decisions.
```

5.5.2.5. Conversions depending on all the decision-categories

In addition to decisions that depend on just one decision category, there are also other decisions that depend on the qualitative decisions made for each of the decision categories. These decisions are the number of pairs to be produced, the number of workers that will be employed and the wholesale price decisions. These decisions depend on all the other decisions and also on the environmental factors: reject rate, worker productivity and the sale amount. These three environmental factors play an important role in the success of a strategy. Therefore, before making these decisions, these factors should be estimated, using the `update_estimations` predicate. According to the following metarule, the estimations are updated after all the qualitative decisions have been converted into their corresponding quantitative decisions.

```
agent : quality_strategy_converted ::= true &
agent : service_strategy_converted ::= true &
agent : advertisement_strategy_converted ::= true &
agent : price_strategy_converted ::= true
==>
{ update_estimations(agent, decision_making) } &
agent : estimations_updated ::= true
--- agent_makes_high_level_decisions.
```

After the environmental factors have been estimated the decisions depending on those estimations can be made. The first is the decision on pairs to be produced. According to that decision, the number of workers employed is established and finally the wholesale price decision is made.

The number of pairs to be produced is based on the sale amount estimation, as well as the reject rate estimation and the inventory left from last year. The amount sale estimation is increased by considering the reject rate estimation and decreased by considering the inventory from the previous year. The following metarule, triggered after the estimation values have been updated, describes how the decision on the number of pairs to be produced is made.

```
agent : estimations_updated ::= true
==>
{ ...,
  Quantity_without_inventory is
            integer(Estimated_Pairs_sold/(1-(Reject/100))),
```

```
  ...,
  New_Quantity is Quantity_without_inventory - Inventory_from_last_year,
  ...} &
agent : number_of_pairs_decided ::= true
--- agent_makes_high_level_decisions.
```

After deciding on the number of pairs, the number of workers required to produce that many pairs is determined by calling the `cal_workers_needed` predicate. This predicate simply divides the number of pairs that will be produced by the estimated worker productivity. The following metarule describes this decision:

```
agent : number_of_pairs_decided ::= true
==>
{ cal_workers_needed(agent, Workers_needed),
  ...} &
agent : number_of_workers_decided ::= true
--- agent_makes_high_level_decisions.
```

The wholesale price decision is made after the number of workers employed decision is made. The price charged for a unit of product is calculated by multiplying the operational cost per product by the coefficient corresponding to the qualitative price category decision. The following metarule describes the process of wholesale price decision-making:

```
agent : number_of_workers_decided ::= true &
{ ... }
==>
{ ...,
  strategy_coefficient( price, Strategy, Coeff ),
  cal_oper_cost_per_produced(agent, Oper_cost),
  New_Price is integer(Oper_cost * Coeff * 100)/100,
  ... } &
agent : price_decided ::= true
--- agent_makes_high_level_decisions.
```

### 5.5.3. LONG-TERM DECISIONS

The plant decisions—automation option and expansion—are associated with the long-term decisions, because the results of those decisions become effective the year after

being decided. Since the effects of those decisions are determined, there is no need for further categorisation in long-term decisions.

The agents make their long-term decisions considering the increase in the market demand. If the demand is estimated to be increasing by a substantial amount, it is to the benefit of the agents to make investments on automation or expansion of their plants in order to increase their plant capacities. If the only concern is to increase the plant capacity, an expansion decision could be made to increase the capacity by a certain amount. If the agent is also concerned with improving the cost of production, an automation decision can also be made.

### 5.5.4. FINANCIAL DECISIONS

All the financial decisions are separated from the short-term and long-term decisions, because these decisions do not have a direct increasing effect on the sale amount of the agents. These decisions affect the market share of an agent indirectly. Agents that are financed by more loans than the others are assumed to make more investments to improve their market share. Although these decisions may have positive impacts on the market share of those agents, they may have difficulty in paying their debts back.

Financial decisions also include the dividend declaration decision that is important for the shareholders of the companies. In order to keep the shareholders pleased, the agents should keep paying dividend. The dividend declaration decision is related to the company's stock price in the market. However, as long as the agents do not fail to pay the dividends, the stock price is not affected. Therefore, the agents should continue paying dividends.

## 5.6. EVALUATION OF THE STRATEGIES

The results of a strategy followed by an agent can be observed by considering the financial indicators of the company managed by that agent. The most important indicators are the Net Income, the ROE (Return on Equity) and EPS (Earnings per Share). The agents who succeed in increasing these indicators are assumed to be successful in managing their companies. The other indicators do not show the success of the strategy because they can be misleading in some situations. The cash indicator for example can be boosted by getting a big amount of short-term loan without affecting the Net Income, ROE and EPS indicators.

The financial indicators of a company can not be calculated without considering the external factors that play a role in the calculation of the financial indicators resulting from the followed strategy. The reject rate, the worker productivity and the sales amount are the external factors affecting the followed strategy. These values are not controlled by the agents and can only be determined at the end of the year in which the selected strategies are followed. Therefore, the evaluation of a strategy follows the determination of those external factors.

As well as the evaluation that occurs at the end of each year, there is also a need for the agents to evaluate their own strategies at the time of their decision-making process. Since the actual values of the external factors can not be calculated before the decisions of all the agents have been gathered together, an agents needs to estimate those external factors. The methods used to calculate the actual and estimated values for the reject rate and the worker productivity are the same, however the methods used to calculate the actual and the estimated sales amount differ due to the lack of information at the time of decision-making.

### 5.6.1. ACTUAL EVALUATION

The actual evaluation of the strategies occurs after all the agents have made their decisions. In the actual evaluation process, the average decisions of all the agents are updated and the reject rate, the worker productivity and the sales amount estimations are replaced with their actual values relative to the other agents' decisions. The financial indicators calculated after the replacement represent the results of the strategies followed by the agents.

#### 5.6.1.1. Actual reject rate

The decisions on the number of models, the quality control budget and the incentive pay per pair are associated with the reject rate that the company will have at the end of the year. The companies that manufacture fewer models, that spend more on quality control budget and that pay more as an incentive per pair to their workers are assumed to have a lower reject rate at the end of the year.

The reject rate of a company is calculated by considering the averages of these decisions that are made by all the agents:

```
Reject_rate is 2 + Models / Actual_Avg_Models +
                   Actual_Avg_Quality_budget / Quality_budget +
                   Actual_Avg_Incentive_pay / Incentive_pay
```

All the companies are assumed to have at least 2% of reject rate no matter what their decisions. Since the number of models is inversely proportional to the reject rate, the effect of the number of models is obtained by dividing it by the average number of models. The effects of other decisions are obtained by simply dividing the average decision by the decision made by the agents.

#### 5.6.1.2. Actual worker productivity

The decisions on the annual wage, the incentive pay per pair, the automation option and the methods improvement budget are associated with worker productivity. The agents that pay more to their workers, that automate their plants and that invest in the methods improvement are assumed to increase their worker productivity.

The worker productivity of a company is calculated as follows:

```
Worker_productivity is (2500 +
                   250 * Annual_wage / Actual_Avg_Annual_wage +
                   250 * Incentive_pay / Actual_Avg_Incentive_pay +
                   Automation_capacity)
                   * Gain_due_methods_improvement
```

Each worker is assumed to have a productivity of 2500 pair per year regardless of other factors. The workers that are paid average are assumed to have 250 pair additional productivity and those who are paid more than the average are assumed to have more than 250 pair; finally, those who are paid less than the average are assumed to have less than 250 pair. As well as the annual wage and the incentive pay of the workers, the automation capacity and the methods improvement budget gain also have an effect on the worker productivity. The former has a fixed impact depending on the option selected and the latter has an impact up to 10% depending on the amount spent per pair. If the amount spent per pair is more than $5, the impact is the highest (10%) and there is no additional impact for the amount above $5.

#### 5.6.1.3. Actual sales amount

The qualitative decisions made for all the decision categories are associated with the sales amount. In order to determine the sales amount of a company, the quantitative decisions

are scored considering the quantitative decisions of all the agents and the market demand is shared among all the companies according to their scores. These scores are arranged so that the average score of all the agents is 1.0. Therefore, the score of each company is 1.0, if all the management teams follow the same strategies.

In order to calculate the strategy score of a company, first of all the decision categories are scored within themselves and the total of these scores is divided by four. For instance, if the quality, service, advertisement and price scores of a company are 1.2, 1.05, 0.90 and 1.03, respectively, the strategy score for that company would be 1.045 and if the average market demand is 2 million pairs per company, this company's sale amount would be 2,090,000 pairs which is 2 million multiplied by 1.045.

The actual sale amount is calculated as follows:

```
Actual_sale_amount is Demand *
                      ( Price_score + Quality_score +
                      Service_score + Advertisement_score ) / 4
```

The score calculation of the decision categories is achieved by calculating the relative scores for the individual decisions associated with those categories. The relative score of a decision is calculated by dividing the individual decision by the average value of that decision made by all the agents, as follows:

```
DecisionX_score is DecisionX / Average_decisionX
```

However, not all the decisions are directly proportional to the results of the company. Some decisions are inversely proportional, therefore the relative score for those decisions is calculated by dividing the average value of that decision by the individual decision of a company, as follows:

```
DecisionY_score is Average_DecisionY / DecisionY
```

The individual scores belonging to one decision category are then added and divided by the number of decisions in that category. The result is the score for that category. For instance, if `DecisionX` and `DecisionY` are associated with a `CategoryZ`, the score for `CategoryZ` is calculated as follows:

```
CategoryZ_score is (DecisionX_score + DecisionY_score) / 2
```

In calculating a decision category score, assuming the same impact of all individual decision scores may result in unrealistic category scores. For instance, a very small difference between the wholesale price to dealers and its average may have a stronger impact in reality than a very big difference more likely to occur between the methods

improvements budget and its average. For instance, if the wholesale price to dealers is 33.0 and the market average is 35.0, this company's score for wholesale price to dealers will be 1.06 because this decision is inversely proportional to the results. However, if the methods improvements budget decision of the same company is $2 million and the average is $1.5 million, the methods improvements budget score will be 1.33, which is much greater than the score for the wholesale price to dealers. The cost of offering to dealers a wholesale price lower than the average price may be much higher than the cost of investing $2 million in the methods improvements budget. However, as a result of considering those two decisions as having the same impact in the category score causes the latter to have a stronger impact on the final decisions. Therefore, there is need to calibrate the individual decision scores according to their impact on the scores of the categories to which they belong.

The `calibration_within_score_categories` predicate is used to define those calibration factors. The sum of the calibration factors for the decisions belonging to the same decision category is arbitrarily arranged to be 1.0, in order to keep the calculation of the category scores simple. For instance, the following two predicates introduce the calibration factors for the `decisionX` and `decisionY` belonging to the `categoryZ`.

```
calibration_within_score_categories(decisionX, 0.7).
calibration_within_score_categories(decisionY, 0.3).
```

These predicates suggests that the `decisionX` has more impact in the `categoryZ` score than the `decisionY`. As a result of having calibration factors, the `categoryZ` score is calculated as follows:

```
CategoryZ_score is DecisionX / Average_decisionX * CalibrationX +
                   Average_decisionY / DecisionY * CalibrationY
```

The calibration factors for the quantitative decisions used in the following subsections have been determined considering each decision's impact on the financial indicators, relative to the other decisions in the same decision category. For instance, we expect to have more annual revenues as a result of the same percentage of change in the price decision than in the method improvement budget decision. This is the reason why the price decision's calibration factor is relatively higher than the methods improvement budget decision's. The use of calibration factors for the decisions provides rational changes in the financial indicators as a result of changes in the decisions.

The relative impact of the decisions within the same decision categories has been determined as a result of the analysis done on the BSG software and it reflects the

author's understanding of the relations between the decisions and their results. These relations have been deduced partly from the BSG software manual and partly by using the BSG software. The BSG software updates the financial indicators immediately after making a change in a decision. The decisions have been practiced with different values and the changes in the financial indicators have been observed to find out the impact of the changes in each decision. The changes resulting from the decisions belonging to the same decision category have been compared and their calibration factors have been determined.

#### 5.6.1.3.1. Price category score

The decisions on the methods improvement budget, the wholesale price to dealers and the customer rebates are used in the calculation of the service category score. The wholesale price to dealers decision is inversely proportional to the price category score. The price category score is calculated as follows:

```
Price_score is
      Methods / Avg_Methods * Calibration_Methods +
      Avg_Price / Price * Calibration_Price +
      Rebates / Avg_Rebates * Calibration_Rebates
```

The calibration factors for the methods improvement budget, the wholesale price to dealers and the customer rebates decisions are 0.3, 0.5 and 0.2, respectively, assuming that the wholesale price to dealers decision is more important than the others.

#### 5.6.1.3.2. Quality category score

The decisions on the percentage of long-wear material, the number of models, the quality control budget, the cumulative quality control budget, the styling/features budget, the annual wages and the incentive pay per pair are used in the calculation of the quality category score. The decision on the number of models is inversely proportional to the quality category score. The quality category score is calculated as follows:

```
Quality_score is
      Long_wear / Avg_Long_wear * Calibration_Long_wear +
      Avg_Models / Models * Calibration_Models +
      Qual_budg / Avg_Qual_budg * Calibration_Qual_budg +
      Cum_qual_budg / Avg_Cum_qual_budg * Calibration_Cum_qual_budg +
      Styling_budg / Avg_Styling_budg * Calibration_Styling_budg +
```

```
Annual_wage / Avg_Annual_wage * Calibration_Annual_wage +
Incentive_pay / Avg_Incentive_pay * Calibration_Incentive_pay
```

The calibration factors for the percentage of long-wear material, the quality control budget, the cumulative quality control budget are defined to be 0.2 and for the other decisions it is defined to be 0.1, assuming that the former have more impact on the quality strategy score than the latter.

#### 5.6.1.3.3. Service category score

The decisions on the number of retail dealers, the number of service representatives and the delivery time are used in the calculation of the service category score. Instead of using the number of service representatives directly in calculating the score, the `ratio` of the number of retail dealers per number of service representatives is used. This `ratio` and the delivery time decision are inversely proportional to the service category score. The service category score is calculated as follows:

```
Service_score is
      Avg_Ratio / Ratio * Calibration_Ratio +
      Dealers / Avg_Dealers * Calibration_Dealers +
      Avg_Delivery_time / Delivery_time * Calibration_Delivery_time
```

The calibration factors for the number of retail dealers per number of service representatives ratio, number of retail dealers and delivery time are 0.3, 0.4 and 0.3, respectively, assuming that the number of retail dealers is more important than the other decisions.

#### 5.6.1.3.4. Advertisement category score

The advertising budget for the current year and the cumulative advertising budget are used in the calculation of the advertisement category score. The advertisement category score is calculated as follows:

```
Advertisement_score is
      Adv_budg / Avg_Adv_budg * Calibration_Adv_budg +
      Cum_Adv_budg / Avg_Cum_Adv_budg * Calibration_Cum_Adv_budg
```

The calibration factors for the advertising budget decision and the cumulative advertising budget are 0.4 and 0.6, respectively, assuming that the latter has more impact on the advertisement category score than the former.

### 5.6.2. ESTIMATED EVALUATION

Besides the actual evaluation process, the agents also need to evaluate their strategies at the time of the decision making process. Since it is not possible to know at that time the actual average decisions, there is a need to estimate the averages. With the assumption that the average decisions of any two consecutive years can not be too different, the calculation of the external factor estimations, except for the sales amount, is done using the previous year's average decisions. The actual average decisions of a year become the estimated average decisions for the following year and the evaluations performed by those estimations are called estimated evaluations.

#### 5.6.2.1. Estimated reject rate

The method used for the estimated reject rate is the same as the one used for the actual reject rate, except for the average values. In the estimation of the reject rate, the previous year's average values are used, as follows:

```
Reject_rate is 2 + Models / Previous_Avg_Models +
                   Previous_Avg_Quality_budget / Quality_budget +
                   Previous_Avg_Incentive_pay / Incentive_pay
```

#### 5.6.2.2. Estimated worker productivity

The method used for the estimated worker productivity is the same as the one used for the actual worker productivity, except for the average values. In the estimation of the worker productivity, the previous year's average values are used, as follows:

```
Worker_productivity is (2500 +
                   250 * Annual_wage / Previous_Avg_Annual_wage +
                   250 * Incentive_pay / Previous_Avg_Incentive_pay +
                   Automation_capacity)
                   * Gain_due_methods_improvement
```

#### 5.6.2.3. Estimated sales amount

At the time of the decision-making process, the estimated sales amount can not be determined using the scoring method for the decision categories, because it is not possible to estimate the score for the price decision category. The cyclic dependency that exist between the sale amount estimation, the wholesale price and the number of pairs to be produced decisions prevents the estimation of the score for that category. The sale amount

estimation depends on the wholesale price decision—the lower the price, the higher sale amount is expected. The wholesale price decision depends on the number of pairs to be produced—the more product, the lower production cost is expected. The decision on the number of pairs to be produced depends on the sale amount estimation—the higher the sale amount estimation, the more pairs need to be produced. As a result of these dependencies, it is necessary to find another method to estimate the sale amount factor.

The sale amount of a certain year is estimated by considering the short-term qualitative decisions and the demand for products in that year. It is assumed that a better qualitative decision yields a better sale amount, therefore each possible qualitative decision (conservative, aggressive and reactive) is associated with a coefficient considering its effects on quantitative decisions. Since the qualitative decisions are ranked according to their results as conservative, aggressive and reactive in increasing order, the coefficients are associated with these qualitative decisions so that the coefficient of a qualitative decision is larger than the coefficient of the preceding qualitative decision. The coefficients for the conservative, aggressive and reactive strategies are 0.0, 0.02 and 0.04, respectively. The coefficients for different possible qualitative decisions are introduced using the following predicate.

```
sale_amount_wrt_strategy(Strategy_value, Coeff).
```

The coefficients are arbitrarily arranged to represent the percentage of the sale amount that will exceed the actual demand. The coefficient corresponding to the conservative decision strategy suggests that the sale amount will be the same as the actual demand. The coefficient corresponding the aggressive decision suggests that the sale amount will be 2% more than the actual demand for each aggressive decision made for a decision category. Finally, the coefficient corresponding the reactive decision suggests that the sale amount will be 4% more than the actual demand for each reactive decision made for a decision category. The sale amount is calculated as follows:

```
Estimated_sale_amount is Demand *
          ( 1 + Price_strategy_coeff + Quality_strategy_coeff +
          Service_strategy_coeff + Advertisement_strategy_coeff )
```

As a result of this method, the sale amount of a company can be estimated and the decision on the number of pairs to be produced can be made considering the estimated sale amount.

## 5.7. THE PROBLEM DECOMPOSITION GRAPH

The managerial decision-making problem is viewed from the student's point of view and that student is represented as the supported agent. The PDG depicts the state of the company managed by the student, as well as its state and objectives. Since the student controls the decision-making process of the supported agent, the objectives and the state can be assumed to be the same as the student's. The main rule in the BSG is named `student`. It is decomposed into several subproblems.

The following is the top-most decomposition rule of the PDG which has six components that are all decomposed further.

```
student <-  state_of_student &
            plant_information &
            history &
            decisions &
            strategy &
            scores.
```

In the rest of this section, the names that appear in *italic* as the arguments of the facts are placeholders and each placeholder name represents the meaning of the argument for which it is used. Further explanation is given only for the arguments for which their placeholder names are not self-explanatory. In the execution of the application, initially all these placeholders are replaced with the values representing the state of the student at the beginning of the game. These values are modified by restructuring metarules at the end of each year so that they always represent the last year state of the student.

### 5.7.1. STATE_OF_STUDENT

The following is the initial decomposition of the `state_of_student` component.

`state_of_student <- making_decision_year_round(`*Year*`,`*Round*`)`.

This decomposition states that the simulation starts with the decision-making process of the student.

- `making_decision_year_round(`*Year*`,`*Round*`)`

  *Year* : the year for which the decisions are made. Initially, this value is set to 10, stating that the game starts with a 10 year-old company.

  *Round* : the number of times the decisions are modified. This argument is used to enable the user to make adjustments in the decisions before they are finalised.

One of the expected modifications in the `state_of_student` component of the PDG occurs as a result of the following restructuring metarule triggered when the student decides to end the decision-making process. After this metarule has been triggered the simulation switches to the `opposing_agents_make_decisions` packet and the state of the student changes to the `waits_for_opposing_agents_decisions`.

```
student : end_decision_making_process ::= true &
student : makes_decision_year_round(Year,_) ::= true &
{ save_all_modifications(student) }
==>
modify ( state_of_student <- waits_for_opposing_teams_decisions(Year) )
switch_to 'opposing_teams_make_high_level_decisions'
--- student_decision_menu.
```

### 5.7.2. PLANT_INFORMATION

The `plant_information` component is decomposed further to represent the facts related to the plant and required in the financial calculations of the company, as follows:

```
plant_information <-
            construction_year_investment_capacity(Year, Cost, Capacity) &
            expansion_status &
            automation_status &
            other_information.
```

The `expansion_status` component is decomposed further as follows:

`expansion_status <- expansions_year_cost_capacity(`*List_of_expansions*`)`.

- `expansions_year_cost_capacity(`*List_of_expansions*`)`

  *List_of_expansions* : a list of triples, each of which represents the year, the cost and the capacity of an expansion decision. For every expansion decision of the student, a new triple is appended to this list.

The `automation_status` component is decomposed further as follows:

`automation_status <- automation_year_option(`*Year, Option*`)`.

The `other_information` component is decomposed further into four components each of which is a fact used in the financial calculations of the company.

```
other_information <- annual_wage_for_last_year(Annual_wage) &
                     cumulative_quality_expenditure(Quality_expend) &
                     cumulative_advertisement_expenditure(Adv_expend) &
                     cumulative_improvement_expenditure(Improve_expend).
```

The expansion_status and automation_status components are decomposed further because the modifications of their facts are conditional. Those components are modified only if the student decides on automation or expansion. However, the facts of the other_information are modified unconditionally. The other_information component is decomposed further in order to avoid replacing the construction_year_investment_capacity fact which is never modified. Therefore, in the rest of the PDG some of the components are decomposed further in order to avoid the unnecessary modifications in the PDG.

The following metarule describes a partial restructuring that occurs in the plant_information component of the BSG, with the use of the procedure construct. This metarule is only triggered if the student makes an automation option selection other than none:

```
student : waits_plant_reseting ::= true &
environment : current_year(Year) ::= true &
{ ...,
  Automation_option \= none }
==>
student : procedure ( automation_status   <-
            automation_option(Year, Automation_option) ) ::= true &
student : waits_plant_reseting ::= false
--- evaluation.
```

### 5.7.3. HISTORY

The history component of the PDG is decomposed further in order to make available the necessary historical financial values to be used in the following years. This decomposition involves some facts from the previous year's results and from the earlier years' results.

```
history <-  last_year_indicators(Revenue, Net_income, EPS, ROE, Cash) &
            last_year_bond_rating(Bond_rating) &
            bonds_issued_so_far(List_of_bond_info) &
```

`short_term_loan_from_last_year(`*Amount_borrowed*`)` &
`shares_and_capital(`*List_of_shares_info*`)` &
`accumulated_retained_earnings(`*Amount_retained*`)` &
`inventory_from_last_year(`*Number_of_pairs*, *Cost*, *Model*, *Quality*`)` .

- `last_year_bond_rating(`*Bond_rating*`)`

  *Bond_rating* : the last year bond rating is determined by considering the debt-to-asset ratio and the times-interest-earned values of the company by using a special formula deduced from the BSG software at the end of each year. Bond rating is used to determine the interest rate that will be applied to the amounts borrowed.

- `bonds_issued_so_far(`*List_of_bond_info*`)`

  *List_of_bond_info* : each element of this list includes the bond number, the amount that is earned by issuing that bond, the amount that is repaid every year and the interest rate that is charged on that repayment. Every time new bonds are issued a new element is appended to this list.

- `shares_and_capital(`*List_of_shares_info*`)`

  *List_of_shares_info* : each element of this list includes the number of shares that is issued and the amount earned by issuing those shares. Every time new shares are issued a new element is appended to this list.

- `accumulated_retained_earnings(`*Amount_retained*`)`

  *Amount_retained* : the earnings of the shareholders that are reinvested, rather than being paid to the shareholders.

- `inventory_from_last_year(`*Number_of_pairs*, *Cost*, *Model*, *Quality*`)`

  *Number_of_pairs* : the number of pairs left in inventory at the end of the previous year.
  *Cost* : per pair manufacturing cost of those products.
  *Model* : the average number of models.
  *Quality* : the average quality rating.

### 5.7.4. DECISIONS

The `decisions` component of the PDG is decomposed to represent the last year's decisions of the student. Those decisions are included as part of the PDG in order to let

the student make his decisions based on his previous year's decisions, rather than starting from scratch.

```
decisions <-    production_decisions(Prod_amount, Use_of_long_wear,
                        Number_of_models, Quality_budg,
                        Styling_budg, Methods_Imp_budg,
                        Reject_rate) &
                labor_decisions(Number_of_workers, Annual_wage,
                        Incentive_pay, Worker_productivity) &
                automation_expansion_decisions(Automation_option,
                        Expansion_capacity ) &
                market_decisions(Wholesale_price, Advertisement_budg,
                        Number_of_dealsers, Rebates,
                        Number_of_sales_rep, Delivery_time,
                        Sale_amount) &
                financial_decisions( Short_term_loan, Share_issued,
                        Bond_issued, Dividend ).
```

Each fact corresponds to a decision category and the arguments of these facts represent the decisions made for the corresponding decision category in the order described in Section 2.1. All these values are updated at the end of each decision-making process of the student. Apart from these decisions, the reject rate, the worker productivity and the sales amount values are also incorporated with the `production_decisions`, `labor_decisions` and `market_decisions` facts, each as the last argument. Those values are modified after all the agents have completed their decisions.

### 5.7.5. STRATEGY

The `strategy` component of the PDG represents the previous year's qualitative decisions that are made for the price, quality, service and advertisement categories of decisions, respectively. This component is updated at the end of each decision-making process of the student and it reflects the objectives of the student in his decisions.

```
strategy    <-    sales_strategy( Price_strategy, Quality_strategy,
                        Service_strategy, Advertisement_strategy ).
```

### 5.7.6. SCORES

The `scores` component of the PDG is decomposed further to represent the scores obtained in the previous year for the price, quality, service and advertisement decision categories, respectively. These scores are modified at the end of the evaluation of all the agents' decisions every year. Since all the agents are assumed to start the simulation with exactly the same state, initially the previous year's scores of all the agents are set to the average score 1.0.

```
scores      <- price_qual_serv_market_scores(Price_score, Quality_score,
                                             Service_score, Advertisement_score).
```

## 5.8. THE DECISION-MAKING PROCESS OF THE OPPOSING AGENTS

The opposing agents are assumed to have exactly the same behaviour in the same situations, therefore their behaviour is described by generalised metarules. In those metarules metafacts are ascribed to the reference `opposing_team` that is defined for the `opposing_management_teams` group.

The decisions of the opposing agents are made autonomously without any user interaction. The decisions of an opposing agent for a decision category are made by considering the previous year's score of that decision category. The long-term decisions are made by considering the change in the market demand estimation. The financial decisions are made considering the financial state of the company and the long-term decisions.

The short-term and long-term decisions definitions are not related, therefore there is no need to provide a certain order in the triggering of the metarules describing those decisions. However, the financial decisions depend on all the quantitative decisions, therefore those metarules need to be triggered after all the qualitative decisions have been converted into their corresponding quantitative decisions.

### 5.8.1. SHORT-TERM QUALITATIVE DECISIONS

Rather than making individually the quantitative decisions affecting a short-term decision category, a new qualitative decision is made for that category, considering the previous year's score for that category, and the new qualitative decision is converted into quantitative decisions by the conversion metarules described in section 5.5.2. The summation of the previous year's score for a decision category and a threshold value defined by the `score_threshold` predicate is compared with the average score for that

decision category, which is arbitrarily arranged to be 1.0 for all the categories. If this summation is less than the average score, then the opposing agent changes the qualitative decision for that category to be reactive. However, if that summation is greater than the average score, then the qualitative decision for that decision category is selected randomly between being aggressive or conservative. This behaviour can be interpreted as follows: the companies that are performing very badly in one decision category need to make an improvement in that category in order to continue their existence in the market, however if they are performing in an average way, the change in that category is not essential, therefore it can be randomly determined.

The selection of being reactive is described by the metarules of the form, in the same way for all the decision categories:

```
opposing_team : defines_strategy ::= true &
{ ...,
  score_threshold(Category, Threshold),
  Score is Category_score + Threshold,
  Score < 1.0 }
==>
opposing_team : short_term_strategy( reactive, Quality,
                                Service, Advertisement ) ::= true &
opposing_team : category_strategy_defined ::= true &
{ ... }
--- opposing_teams_make_high_level_decisions.
```

*Category* and *category* should be replaced by an actual category name and the corresponding parameter of that category, in the `short_term_strategy` predicate, should be given the constant value `reactive`. In this metarule, the `reactive` value is given to the first parameter used for the price category, assuming that the metarule defines the qualitative price category decision.

The selection of being conservative or aggressive is described by the metarules of the form, in the same way for all the decision categories:

```
opposing_team : defines_strategy ::= true  &
{ ...,
  score_threshold(Category, Threshold),
  Score is Category_score + Threshold,
  Score >= 1.0,
```

```
  random_member(New_Category_Stgy, [conservative, aggressive]) }
==>
opposing_team : short_term_strategy( New_Category_Stgy, Quality,
                                     Service, Advertisement ) ::= true &
opposing_team : category_strategy_defined ::= true &
{ ... }
--- opposing_teams_make_high_level_decisions.
```

*Category* and *category* should be replaced by an actual category name. In the determination of the new qualitative decision the `random_member` predicate is used and the result of that predicate is given to the corresponding parameter of that category in the `short_term_strategy` predicate. In this, the `New_Category_Stgy`, the result of the `random_member` predicate, is given to the first parameter used for the price category, assuming that the metarule defines the qualitative price category decision.

### 5.8.2. LONG-TERM DECISIONS

The automation option decision of the opposing agents is made randomly, because all the options have an improving effect on the process of manufacturing and in the production cost. In order to obtain the various decisions that are likely be made by different management teams in reality, this decision is made without considering any environmental factors. The following metarule describes the random selection of the automation option decision for the opposing agents.

```
opposing_team : defines_strategy ::= true &
{ ...,
  Existing_option = none,
  random_member(New_option, [ none, a, b, c, d ]),
  ... }
==>
...
--- opposing_teams_make_high_level_decisions.
```

This metarule is triggered if an automation option decision has not yet been made. As a result of this metarule, the new automation option is ordered on behalf of an opposing agent and it becomes effective in the following year.

The expansion decision is made by considering the market demand for the following year. In order to find the market demand estimation for the following year, the market demand

estimation for the year for which the decisions are made is multiplied by a factor which is randomly determined within a certain range. This reflects the variance expected to arise in the estimations of different agents. As a result, there will be differences in the expansion decision making process of the opposing agents. The `production_capacity` predicate returns the capacity of the plant managed by a certain agent for comparison with the estimated demand. If the current capacity of the plant does not satisfy the demand estimation for the following year, the opposing agents are assumed to make an expansion decision for their plant. The following metarule describes the expansion decision process.

```
opposing_team : defines_strategy ::= true &
( existing_metafact( environment, demand(Demand_per_company), true ),
  production_capacity( opposing_team, Capacity_of_plant),
  random(100,110,Demand_Inc_estimation),
  Demand_estimation is Demand_per_company * Demand_Inc_estimation / 100,
  Capacity_of_plant < Demand_estimation,
  random_member(New_capacity_increase, [1000,2000,3000]),
  ... )
==>
{ ... }
--- opposing_teams_make_high_level_decisions.
```

The above metarule is triggered only if the plant capacity does not meet the demand in the environment for the products. If this happens, the expansion in the capacity of the plant is achieved randomly from the pre-defined choices.

### 5.8.3. FINANCIAL DECISIONS

The financial decisions depend on the investment decisions made this year and on the decisions that are made last year to be financed this year. The former decisions are the investments in quality control, in methods improvements and some others that become effective immediately. The latter are the automation option and expansion decisions that are long-term decisions. The cost of the long-term decisions are paid in the year they become effective.

The financial decisions are considered from two point of views. One of them is the decisions that need to be made because of the long-term decisions made in the previous year. The other are the decisions that need to be made because of this year's investments that will be effective immediately. The lack of cash that is due to the long-term decisions is met by long-term financing methods: issuing new shares or new bonds. The lack of

cash that is due to the short-term investments is met by a short-term financing method: the request for a short-term loan from a financial institution.

The short-term financing decisions depend on the decisions made for the current year, therefore they need to be made after the last quantitative decisions have been made, which are the price category decisions. The long-term financing decisions depend only on the decisions that are made in the previous year. Therefore the amounts that need to be financed due to those decisions can be determined anytime without any restriction. However, the selection of the financing method should be done after these amounts have been determined. In order to satisfy this, the long-term financing decisions are also made at the same time as the short-term financing decisions.

#### 5.8.3.1. The long-term financial decisions

The following metarule describes the determination of the financing amount for the automation option cost. This metarule is triggered only if the automation option decision has been made in the previous year. If so, the fixed cost of the automation option is determined by using the `auto_cost_setup_prod` predicate and that value is added to the long term financing amount (it is 0 at the beginning of every year), to be used later.

```
opposing_team : defines_strategy ::= true &
{ existing_metafact(environment, current_year(Current_year), true ),
  existing_metafact(opposing_team,automation_option(Year,Option),true),
  Year_difference is Current_year - Year,
  Year_difference = 1 }
==>
{ auto_cost_setup_prod(Option,Investment_amount,_,_),
  existing_metafact(opposing_team,
                    long_term_financing_amount(Old_amount), true),
  New_amount is Investment_amount + Old_amount } &
opposing_team : long_term_financing_amount(New_amount) ::= true
 --- opposing_teams_make_high_level_decisions.
```

The following metarule describes the determination of the financing amount for the expansion decision cost. This metarule is triggered only if an expansion decision has been made in the previous year. The `last_year_expansion_decision` predicate is used to determine whether an expansion decision in the previous year has been made. If not, this predicate fails, otherwise the fixed cost of that expansion decision is returned as a result of that predicate. That value is also added to the long term financing amount for later use.

```
opposing_team : defines_strategy ::= true &
{ last_year_expansion_decision( opposing_team, Expansion_amount ) }
==>
{ existing_metafact(opposing_team,
                    long_term_financing_amount(Old_amount), true),
  New_amount is Expansion_amount + Old_amount } &
opposing_team : long_term_financing_amount(New_amount) ::= true
--- opposing_teams_make_high_level_decisions.
```

The following metarule describes the selection process of the long-term financing method. This metarule is triggered only if the long term financing amount is not zero (this happens when there have been no long-term decision in the previous year). If the long term financing amount is not zero, a long-term financing method is selected randomly from the new share and new bond issuing methods. According to the financing method selected randomly, the financing is made by using the `long_term_finance_decision` predicate, considering the amount that needs to be financed. Using the randomly determined value of the `Financing_method`, this predicate simulates issuing as many new bonds or new shares as necessary, in order to finance the amount of `Amount_to_be_financed`.

```
opposing_team : price_decided ::= true &
{ existing_metafact(opposing_team,
                                long_term_financing_amount(Amount), true),
  Amount > 0,
  debt_to_asset_ratio(opposing_team, Ratio),
  Amount_to_be_financed is Amount * Ratio,
  random_member(Financing_method,[new_bond_issuing,new_share_issuing]),
  long_term_finance_decision(opposing_team,
                                Financing_method, Amount_to_be_financed),
  net_income(opposing_team, Net_income),
  shares_out(opposing_team, Shares_out, _ ),
  New_dividends is ( Net_income - (Amount / Ratio) ) / Shares_out,
  ... }
==>
{ ... } &
opposing_team : long_term_financial_decisions_made ::= true
--- opposing_teams_make_high_level_decisions.
```

Not all amounts require long-term financing method, because the management teams are assumed to follow the residual dividend policy (Ross *et al.*, 1993). According to this policy, the teams try to keep a constant debt to asset ratio. In order to achieve this, only the amount proportional to the debt to asset ratio receives long-term financing and the rest is financed by the company's net income. The `debt_to_asset_ratio` predicate returns the ratio which is multiplied by `Amount` to calculate the required long-term financing. The net income, calculated by the `net_income` predicate, that is not spent on the long-term investment is shared by the shareholders as dividends.

#### 5.8.3.2. The short-term financial decision

The short-term financial decision is made by simply considering the estimated cash value calculated after all the quantitative decisions have been made on the basis of the estimated average decisions. Since the long-term financial decisions also affect the cash value of the company, this decision should be made after the long-term financial decisions. The cash value is determined by `cash` predicate, that requires the revenues value of the company that is calculated by the `revenues` predicate. The following metarule is triggered only if the cash value is negative as a result of all the quantitative decisions. If so, a request for a short-term loan decision is made to achieve a positive cash balance.

```
opposing_team : price_decided ::= true &
opposing_team : long_term_financial_decisions_made ::= true &
{ revenues( opposing_team, Revenues),
  cash( opposing_team, Revenues, Cash),
  Cash < 0,
  New_short_term is -Cash }
==>
{ ... }
--- opposing_teams_make_high_level_decisions.
```

## 5.9. DECISION-MAKING PROCESS OF THE SUPPORTED AGENT

The control of the decision-making process of the supported agent is given to a student, so that he can practice his theoretical knowledge. In the BSG application of Negoplan, the user is also given qualitative decision-making ability which is believed to happen more likely in real-life, as well as being given quantitative decision making ability, which is provided in the BSG software. In the decision-making process, the student is free to

choose any of these two decision-making processes. It is also possible to switch between those two processes while making decisions for a certain year.

The qualitative decisions are converted to their corresponding quantitative decisions by the metarules defined in section 5.5.2. These metarules are used for both the student and the opposing agents. Since these metarules may not come up with the exact decisions that the student was expecting, he may want to make adjustments in the quantitative decisions that are resulted from those metarules. Therefore, the real intention of providing both of these decision-making processes to the student is to enable him to make changes when the decisions made by the conversion metarules are not satisfying.

Both the qualitative and the quantitative decision-making processes proceed within cycles until the student decides to stop. These cycles give the student a chance to modify his qualitative decisions and his quantitative decisions until the results obtained are satisfying.

In the quantitative decision making process, the student is asked to make production, labour, plant, marketing and financial decisions, one at a time, in an order specified by the student. After making decisions of a group, the financial indicators are calculated by considering the modifications in the decisions and the estimated average decisions, as well as some other intermediate indicators that are related to the decisions of that group and used in the calculation. All of these indicators are displayed on the screen with that group's decisions, so that the student can make observations on the possible impact of his decisions.

In the qualitative decision making process the student is expected to make short-term qualitative decisions for the price, quality, service and advertisement categories, long-term decisions and finally financial decisions. These decisions are exactly the same as the decisions made by the opposing agents. The short-term qualitative decision-making involves selection among being conservative, aggressive and reactive, for each of the decision categories. The long-term decision-making involves the selection of an automation option and an expansion decision. The financial decision-making involves decisions related to short-term and long-term financing methods. After all these decisions have been made, the qualitative decisions for the decision categories are converted into their corresponding quantitative decisions and the financial indicators are calculated by considering these decisions and the estimated average decisions. Since the qualitative decisions relate to all the decision groups that appear in the quantitative decision-making, the screens that are displayed to the user after modifying each of those groups are shown altogether at the end of the qualitative decision-making cycle.

The following section introduces the decision screens displayed after quantitative and qualitative decision-making cycles, in order to give the student an idea about the results that will be obtained as a result of his decisions. The decision screen corresponding to a group of decisions is displayed alone after making a change in that group of decisions. All the decision screens are displayed after a qualitative decision-making process, in order to show the user the estimated results of his decisions.

The quantitative values shown in the following screens are the set of decisions made by the previous management team in the last year before starting the game. All the financial indicators and the intermediate indicators that are related to the decisions appearing on a particular screen and used in the calculation of the financial indicators are calculated considering the estimations of the reject rate, worker productivity and the sales amount of the company. Therefore, these financial indicators are the estimated values and the actual financial indicators are calculated after all the agents have made their decisions. Those indicators appear directly in the PDG of the student.

### 5.9.1. PRODUCTION DECISIONS SCREEN

| | | |
|---|---|---|
| Pairs to be Manufactured (000s) | | 1989 |
| Percent Use of Long-Wear (0-100) | | 25 |
| Number of Models (50, 100, 200) | | 100 |
| Budgets: ($ 000s) | Quality Control | 785 |
| | Styling/Features | 500 |
| | Method Improvements | 0 |
| Reject Rate Estimation | | 5.0 |
| Quality Rating of Pairs Produced | | 111 |
| Total Manufacturing Cost ($000s) | | 36097 |
| Mfg Cost Per Pair Produced ($) | | 19.1 |

Revenues:56164 Net Income:7166 EPS:1.43 ROE:16.86 Cash:258

### 5.9.2. LABOR DECISIONS SCREEN

| | |
|---|---|
| Number of Workers employed | 663 |
| Annual Wages (000s of $) | 12.0 |
| Incentive Pay ($/pair) | 1.5 |
| What IF : Worker Productivity | 3000 |
| Number of Workers Needed | 663 |
| Percent Change in Annual Wage | 0.0% |
| Total Pay Per Worker (000s) | 16.2 |

```
Incentive Pay as % of Total Pay     25.9%

Labor Cost Per Pair Produced ($)     5.4
Mfg Cost Per Pair Produced ($)      19.1

Revenues:56164    Net Income:7166     EPS:1.43     ROE:16.86    Cash:258
```

### 5.9.3. AUTOMATION AND NEW CAPACITY DECISIONS SCREEN

```
-------------------------------------------------
Automation Option (A,B,C,D or N)              none
Expansion of Existing Plant (000s of prs)        0
-------------------------------------------------

Cash Outflows for Capital Expenditures:
    Current Year=       0     Next Year=       0

Revenues:56164    Net Income:7166     EPS:1.43     ROE:16.86    Cash:258
```

### 5.9.4. MARKETING DECISIONS SCREEN

```
-------------------------------------------------
Wholesale Price to Dealers ($/pair)          31.22
Advertising Budget (000s of $)                2000
Number of Retail Dealers                       500
Customer Rebates (0, 1, 3, or 5 $)               3
Customer Service Staff (total number)           10
Delivery Time (4, 3, 2, or 1 week)               3
-------------------------------------------------
Sales amount estimation (000s of pairs)      1799
-------------------------------------------------
Pairs available for Sale (000s)              1889

Marketing Expenses ($/pair sold)             2.35
Operating Costs ($/pair sold)               24.18
Operating Profit ($/pair sold)               7.04
Total Operating Profit (000s of $)          12664

Revenues:56164    Net Income:7166     EPS:1.43     ROE:16.86    Cash:258
```

### 5.9.5. FINANCIAL DECISIONS SCREEN

```
-------------------------------------------------
Request for Short-Term Loan (000s of $)          0

Shares Issued (000s)                             0

Bond Issue (000s of $)                           0

Dividend Declaration ($ per share)             0.4
-------------------------------------------------

Debt To Asset Ratio:0.51   Common Shares outstanding:5000    for $15.0
Times_Interest_Earned:5.22   Projected Bond Rating:BBB

Revenues:56164    Net Income:7166     EPS:1.43     ROE:16.86    Cash:258
```

## 5.10. RESETTING THE ENVIRONMENT

After the decisions of all the agents have been made and evaluated for a particular year, the game is expected to proceed with the decision-making process for the following year. However, between any two consecutive years, there may be some changes in the environmental facts. These facts are modified after the evaluation of each year's decisions by considering the decisions made in that year and the changes occurring in the state of the companies as a result of the evaluations, randomly within certain ranges.

The environmental facts, long wear and normal wear material prices are modified by considering the long-wear decisions that have just been evaluated. The long wear material price is expected to rise when the average percentage use of long wear material exceeds a certain percentage. As a result of this, the normal wear material price is expected to decrease because these materials complement each other. The percentage considered as the threshold is defined by the `long_wear_percentage_threshold` predicate. If the average percentage is above the threshold value, the modifications in the long wear and in the normal wear prices are determined randomly within certain ranges. The following metarule describes the behaviour of material price adjustments occurring in the market depending on the average decisions.

```
environment : long_wear_price(Long_price) ::= true &
environment : normal_wear_price(Normal_price) ::= true &
{ ...,
  long_wear_percentage_threshold(Threshold_long_percent),
  Avg_long_percent > Threshold_long_percent }
==>
{ random(0,10,Long_increase_percent),
  random(0,10,Normal_decrease_percent),
  Long_factor is 1 + Long_increase_percent / 100,
  Normal_factor is 1 - Normal_decrease_percent / 100,
  New_Long_price is integer(Long_price * Long_factor * 10)/10,
  New_Normal_price is integer(Normal_price * Normal_factor * 10)/10 } &
environment : long_wear_price(New_Long_price) ::= true &
environment : normal_wear_price(New_Normal_price) ::= true
--- environment_resetting.
```

There is no need to consider the situation when the long wear material percentage decreases, because the agents are assumed always to make improving decisions.

The environmental facts concerning stock prices for each company are modified by considering their financial states after the evaluations. The financial indicators taken into consideration in the calculation of the new stock price are the ROE and EPS indicators. These indicators are calculated by the `roe` and `eps` predicates which require the net income as input. The stock price of a company is expected to rise when these indicators for that company are above the averages of all the companies and is expected to decrease when they are below the averages. The change in the stock price is indexed to the average of the percentages between those indicators and the averages of all the companies. The following metarule describes how the stock prices of each company are adjusted according to its financial indicators. In order to avoid presenting here the same metarule once for the student and once for the opposing agents, the metafacts that need to be ascribed to agents are ascribed to `agent`.

```
agent : stock_price_modification ::= true &
environment : stock_price(agent,Price) ::= true &
{ net_income( agent, Net_income)
  roe( agent, Net_income, Roe ),
  eps( agent, Net_income, Eps ),
  calculate_average_roe( Avg_Roe ),
  calculate_average_eps( Avg_Eps ),
  Stock_price_factor is ( Roe / Avg_Roe + Eps / Avg_Eps ) / 2,
  New_Price is integer(Price * Stock_price_factor * 10) / 10 } &
environment : stock_price(agent,New_Price) ::= true
--- environment_resetting.
```

For example, if the ROE and EPS indicators of a company are 16.5 and 1.5 respectively and if the averages of those indicators of all the companies are 15.0 and 1.6 respectively, the stock price changes from $15.0 to $15.28.

The modifications of the environmental metafacts simulate the changes that occur in reality, and the fact that those modifications are autonomous makes the BSG application of Negoplan more realistic.

## 5.11. Review of the BSG application

The categorisation of the quantitative decisions according to their mutual relevance has played a very important role in the implementation of the BSG application of Negoplan. The major obstacle in this application was the creation of opposing agents autonomously

adapting their decisions according to the changing environment and to their state. This capability of the opposing agents is achieved by categorisation.

The definition of some possible qualitative decisions for the decision categories reduced the number of alternative decisions substantially and the conversion mechanism presented in section 5.5.2 enabled the quantitative decisions of the opposing agents to be generated automatically according to their qualitative decisions for different categories. The categorisation and the definition of qualitative decisions also enabled the student not to deal with quantitative decisions and thus an improvement has been achieved over the BSG software.

The scoring method for the decision categories presented in section 5.6 is used in the determination of the market share of each company, which is the key factor in the evaluation of the decisions. The scoring is also used by the opposing agents in the determination of an adaptive qualitative decision for each decision category.

The BSG application that involves two opposing agents is implemented on Negoplan using its new extensions presented in Chapter 4.

# 6. OBSERVATIONS AND EXPERIMENTAL RESULTS

This chapter presents the observations made in the implementation phase of the BSG application and the results of the experiments.

## 6.1. OBSERVATIONS ON THE BSG APPLICATION OF NEGOPLAN

The Negoplan system is used to implement applications for cause-effect type problems. Such problems include decision-making problems that are sequential and context-dependent. A Negoplan application for a specific problem involves three main steps: analysis of the problem, translation of the problem objectives and the cause-effect type relations into Negoplan constructs, and coding the definitions that are problem-specific. The first and third steps, very general, would appear almost in any application in any other language. However, Negoplan constructs play a very important role in the second step and allow one to create a running simulation without requiring the programming knowledge necessary in other languages.

The Negoplan system provides all the constructs that someone may require to develop a decision-making problem simulation. In real life, the participants in a decision-making problem always consider it from their point of view and make decisions in keeping with their own objectives and internal states. The Problem Decomposition Graph (PDG) construct in Negoplan works for simulations that can be developed from only one participant's point of view and is used to describe the objectives and internal state of that participant without getting involved with any data structure detail. This is the student team in the BSG application. The metafacts are used to represent the instantaneous states of the participants that are subject to change as a result of the interaction described by response metarules between the participants. The changes in the instantaneous states of the participants may result in modifications described by the restructuring metarules in the PDG.

Negoplan metarules enable one to describe a situation as a chain of cause-effect relations, traversed by a built-in forward chaining engine. When implementing an application, the only concern is to construct an unbroken chain of cause-effect relations, that is, to ensure that each relation is likely to occur. Once this has been achieved, those relations are processed by the forward chaining engine. This allows one to focus on the determination and the ordering of the relations, rather than how they will be implemented (as it might be necessary in other languages). In the BSG application, since the decision-making process of the agents requires making decisions in a certain order, the availability of these

constructs was very effective in the description of the relations between different decisions.

The Negoplan system's original intention is to develop qualitative rather than quantitative simulations. However, the availability of Prolog procedure calls embedded in metarules enabled the BSG application that is highly quantitative. Providing the means to develop quantitative simulations does not mean that the application of such problems should be implemented by someone who is both experienced in the problem's field and familiar with Prolog. Such problems can easily be partitioned so that the quantitative coding of the problem-specific definitions and the qualitative description of the problem simulation can be implemented by more than one person. The simulations can be built using metarules and the flow of the simulations can be verified without the availability of the quantitative elements. The two parts can be put together when they are ready. Although this was not an important issue in the implementation of the BSG application, because both parts were implemented by one person, it was still an asset: once all the quantitative problem-specific definitions were ready, the focus was completely given to the development of the problem simulation.

As a result of all these properties, in the BSG application the time was mostly spent on the improvement of the simulation, rather than on its implementation.

## 6.2. Experimental Results

The BSG application has been implemented in Negoplan. The main issue in testing the application was to validate the autonomous behaviour of the opposing agents. This has been achieved by experiments described in section 6.2.1. Besides this, different qualitative decisions have also been experimented with and their results are explained in section 6.2.2. The validation of the implementation is based on two indicators, `revenues` and `net income`, calculated as follows:

```
revenues = sale amount * price
net income = operational profit - interest - tax
```

where

```
operational profit = revenues - operational cost
```

The company is considered growing if its `revenues` and `net income` increase. Increasing `revenues` alone can be a misleading indication if it does not result in higher `net income` at the same time.

### 6.2.1. THE RESULTS AGAINST THE OPPOSING MANAGEMENT TEAMS

The results of the student against the opposing agents are observed from two perspectives. The qualitative decisions of the student are conservative against opposing agents that adapt their strategies according to the changing conditions; they are aggressive against all conservative opposing agents. The experiments are run several years until a certain trend appears in the indicators of the companies.

#### 6.2.1.1. Conservative student vs. autonomous opposing agents

In this experiment, the opposing agents are allowed to make their decisions autonomously according to the behaviour described by metarules presented in Section 5.5. The simulation is run for five years.

Since the conversion methods are based on the previous years' average decisions, the differences between the quantitative decisions of the agents never exceed the limits that require being reactive. Thus, in the five-year period no opposing agent needed to be reactive. Being conservative or aggressive is determined randomly. Despite the student's conservative decisions, his actual revenues increase too, because the conservative decisions are converted into the quantitative decisions by replacing them with the previous year's average decisions. Since the quantitative decisions always increase in order to get more of the market demand, the average decisions also show an increasing trend and this results in the conservative agents increasing their investments, too.

The short-term and long-term decisions of the opposing agents are as follows, for the five-year period:

| | | Short-term Strategy | | | | Long-term Strategy | |
|---|---|---|---|---|---|---|---|
| | | Price | Quality | Service | Advertisement | Automation | Expansion |
| 1 | Agent1 | aggr. | aggr. | cons. | aggr. | A | |
| | Agent2 | cons. | aggr. | aggr. | aggr. | | |
| 2 | Agent1 | cons. | cons. | aggr. | aggr. | | 1000 |
| | Agent2 | aggr. | aggr. | aggr. | cons. | | 3000 |
| 3 | Agent1 | aggr. | cons. | aggr. | cons. | | |
| | Agent2 | aggr. | cons. | cons. | aggr. | B | |
| 4 | Agent1 | aggr. | cons. | cons. | cons. | | |
| | Agent2 | aggr. | aggr. | aggr. | cons. | | |
| 5 | Agent1 | aggr. | aggr. | aggr. | aggr. | | |
| | Agent2 | cons. | cons. | aggr. | cons. | | |

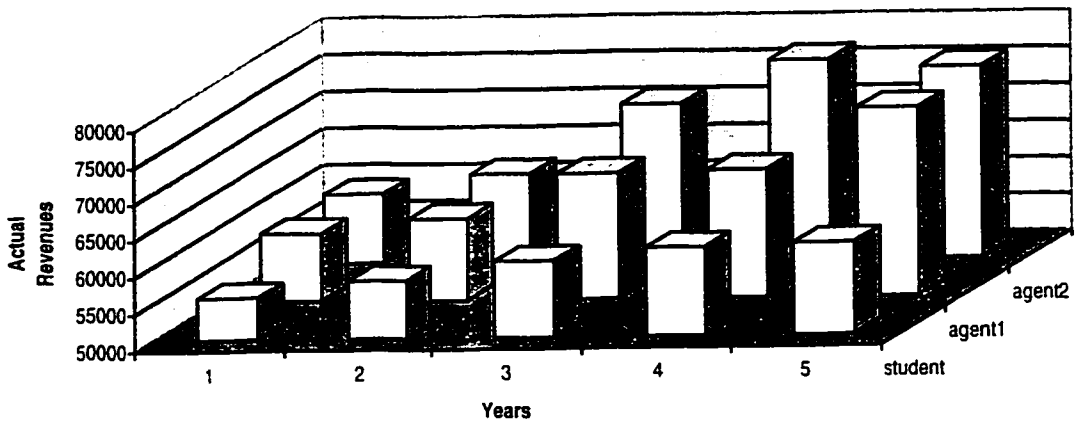



Figure 5. Actual Revenues (autonomous opposing agents)

The agents' actual revenues obtained as a result of these qualitative decisions, after the evaluation at the end of each year, are presented in Figure 5. The actual revenues of the opposing agent 1 and the opposing agent 2 increase more than the student's actual revenues, as expected. The actual revenues of opposing agent 2 increases more than those of opposing agent 1 in the first 4 years because of its better strategy in year 2. At the end of year 5 they become very close because opposing agent 2 makes more conservative decisions than opposing agent 1.

The Figure 6 shows the actual net incomes of the agents as a result of the evaluations performed at the end of each year. The net incomes do not have the same tendency as the revenues because of the long-term decisions made. The student's net income increases because no long-term decision is made by the student. However, the net income of the opposing agents fluctuate because of their long-term decisions.

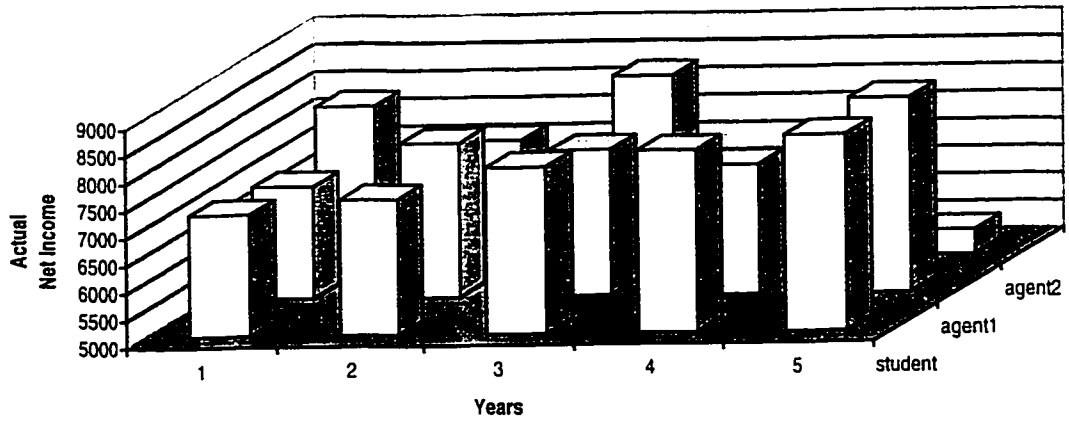



Figure 6. Actual Net Incomes (autonomous opposing agents)

The opposing agent 1 was aggressive in price and quality. This has been shown to be less effective than being aggressive in service and advertisement—see section 6.2.2. That's why the opposing agent 2 has a higher net income than the opposing agent 1.

The net income of the opposing agent 1 increases more than the others' income at the end of the year 2, because of the automation option that became available in year 2. However, at the end of the year 3 and 4, there has been a decrease in the net income of opposing agent 1, because of the overload due to financing the automation investment and the expansion decision (1000 pairs) in year 2. At the end of the year 5, the net income of opposing agent 1 starts to increase again.

The net income of the opposing agent 2 decreases sharply at the end of the year 2 because of being aggressive in price, not very effective for net income. At the end of the year 3, its net income increases sharply as a result of the expansion decision (3000 pairs) that became available in year 3. However, at the end of the year 4 and 5, it decreases drastically because the amount borrowed to finance the expansion decision is beginning to be paid back in year 4. The decrease is higher than the decrease in the net income of the opposing agent 1 for his expansion decision. That's because opposing agent 2 decided on the expansion to 3000 pairs which is more expensive than 1000 pairs.

#### 6.2.1.2. Aggressive student vs. conservative opposing agents

In this experiment, the short-term qualitative decisions of the opposing agents are conservative, but they are allowed to make long-term decisions autonomously. The student is forced to be aggressive in all decision categories and to make the automation decision and the expansion decision ( by 1000 pairs) in the year 1. These short-term and long-term strategies are selected with the expectation for the student to have better results than the opposing agents. Since we know from the previous experiment that revenues and net income both gradually increase for pure conservative strategies, the opposing agents are allowed to make long-term decisions to enable us to make more observations in the results of those decisions. The simulation is run for six years. The opposing agent 1 made an automation decision in year 1 and an expansion decision (2000 pairs) in year 4. The opposing agent 2 made an automation and expansion decisions (1000 pairs) in year 3.

Figure 7 depicts the actual revenues of the agents at the end of each year. The revenues of the *aggressive* student increased more than of the opposing agents. The actual revenues of the conservative opposing agents show also an increase because the average decisions tend to increase, as a result of having agents that always consider the ways of making new improvements.

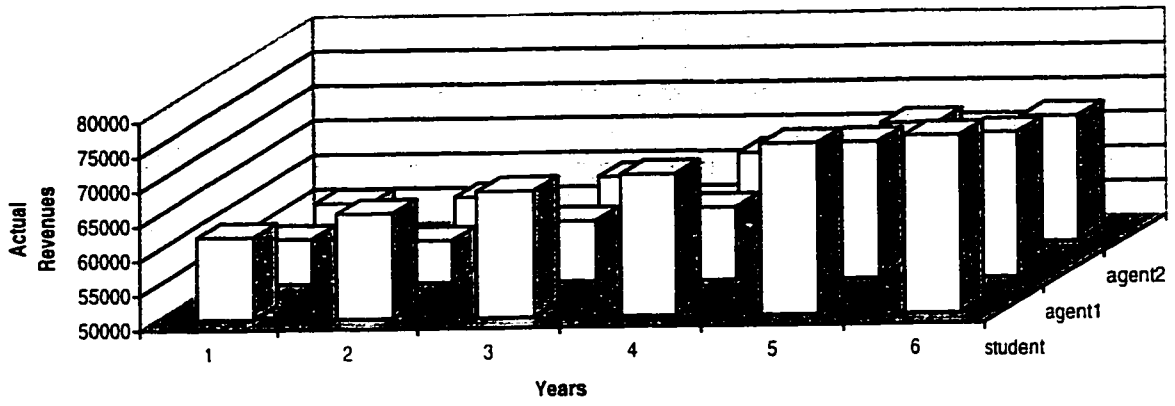



Figure 7. Actual Revenues (conservative opposing agents)

Figure 8 depicts the actual net incomes of the agents at the end of each year. The net income of the student decreases after year 3, because of the overload caused by the repayment of the long-term financing decisions made in year 2, to finance the automation and expansion decisions made in year 1. After the year 3, the net income of the student begins to increase again as expected. Since the student is aggressive, its actual net incomes are always bigger than the opposing agents' actual net incomes.

The net income of the opposing agent 1 increased at the end of the year 5, as a result of the expansion decision made in the year 4. However, this change in the net income is temporary, only until the repayments of the financing start, in year 6. The decrease in the student's net income in the year 3 was not as sharp as the decrease for opposing agent 1, because the student was aggressive while the opposing agent 1 was conservative.

The same trend can also be observed in the net incomes of the opposing agent 2. After making the expansion decision in year 3, the net income increased in year 4, but in year 5 it decreased again.

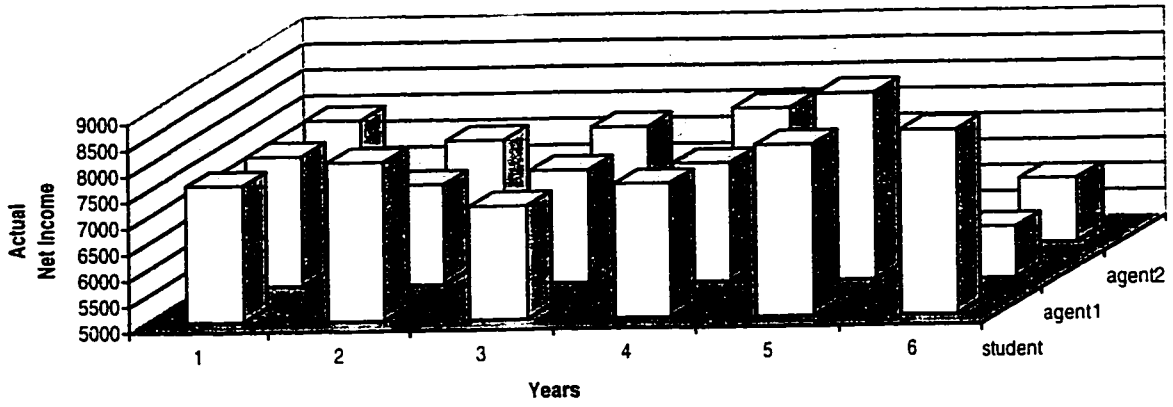



Figure 8. Actual Net Incomes (conservative opposing agents)

### 6.2.2. THE RESULTS OF DIFFERENT QUALITATIVE DECISIONS

The quantitative decisions are collected in several decision categories in order to enable qualitative autonomous decision-making process of the opposing agents. The categories are `price`, `quality`, `service` and `advertisement`, and the qualitative decisions that can be made for these categories are `conservative`, `aggressive` and `reactive`. The estimated results of the student's first year decisions are observed with different qualitative decisions in order to validate the results of these qualitative decisions. In order to visualise these results in a three-dimensional graph, the data have been collected by making the same qualitative decisions for the pairs of decision categories, rather than to individual categories. Tests with all different combinations of pairs gave similar results. Therefore, the results of the price-quality and the service-advertisement pairings have been arbitrarily chosen to represent all the other possible pairings and used in the following discussion.

As a result of the methods used in the conversion of qualitative decisions into quantitative decisions, described in section 5.5.2, the conservative agents are expected to have lower revenues and lower net income than other agents.

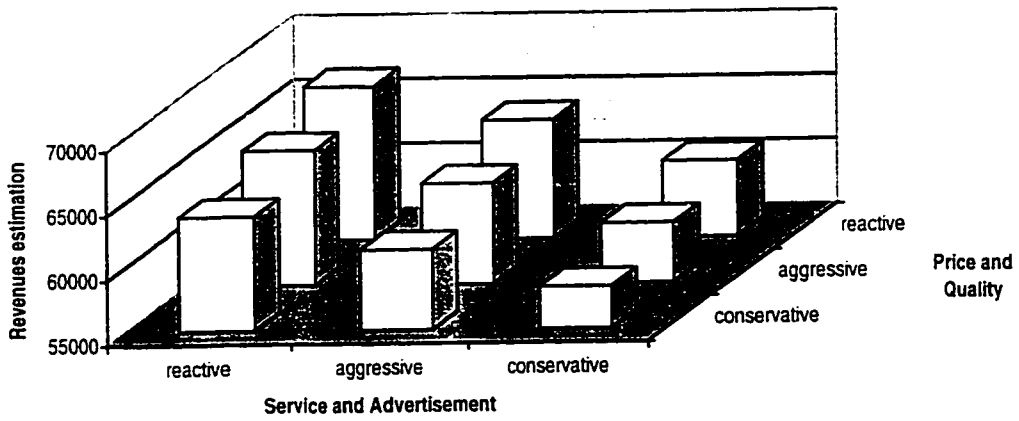



Figure 9. Revenues for different qualitative decisions

Figure 9 depicts the estimated revenues for all combination of price-quality and service-advertisement qualitative decisions in the first year decision-making process of the student. The graph shows that the change in the revenues is as expected: an increase when the qualitative decisions made changes from conservative to aggressive and aggressive to reactive. The graph also shows that the revenues tend to be higher when the service-advertisement qualitative decisions are less conservative than the price-quality qualitative decisions. For instance, when the price-quality and service-advertisement qualitative decisions are reactive and conservative, the revenue estimation is more than $60,000 but

it is lower than $60,000 when they are conservative and reactive, respectively. This difference suggests that the BSG application simulates a market demand towards well advertised and well serviced products rather than low price and high quality products.

Figure 10 depicts the estimated net incomes corresponding to the revenues for the same qualitative decisions presented in Figure 9. An expected increase appears, as expected, in the net income for service and advertisement qualitative decisions changing from conservative to reactive. The net income was also expected to increase the same way for price and quality qualitative decisions, but it decreases although the same qualitative decisions result in increasing revenues. This unexpected result in the net income suggests that either the price and quality qualitative decisions do not have enough increasing impact in the sale amount or the price qualitative decision results in a very low sale price.

In fact, the decrease in the net income is an expected result according to the scoring method used to evaluate the strategies, but not realistic. The changes in external factors depend on the ratio between the decisions and the average decisions. A 3% decrease in price is expected to cause a 3% increase in the sale amount—note that it is an external factor. However, since a 3% price decrease means a much higher profit decrease, this loss is not compensated by a 3% increase in the sale amount.

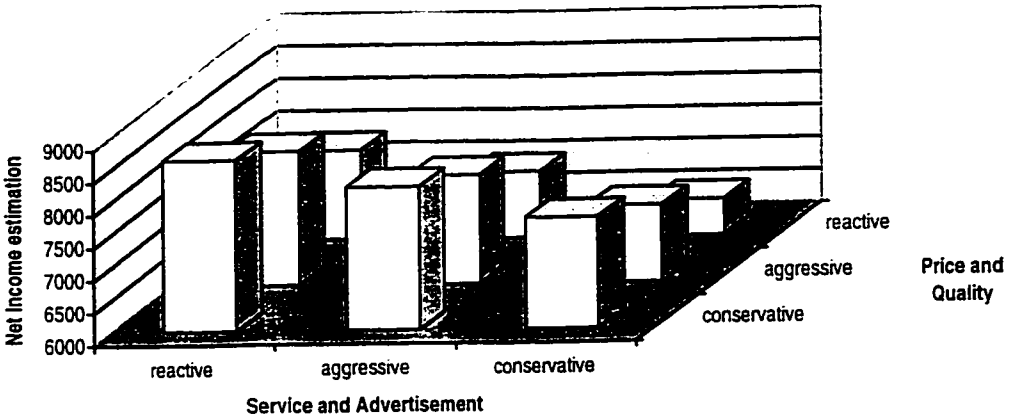



Figure 10. Net Income for different qualitative decisions

### 6.2.3. Conclusions

In the experiments discussed in section 6.2.1, it is observed in general that the net income of a company increases in the year succeeding a plant automation or plant expansion decision and decreases in the year succeeding a long-term financing decision. The impact of these decisions on the net income depends on the short-term qualitative decisions made. For example, for an aggressive agent the impact of long-term financing decision is

weaker than for a conservative one, resulting in smaller decrease. After having these temporary impacts of long-term decisions, the net income continues to change as a result of the short-term qualitative decisions.

As a result of the experiments presented in section 6.2.2, the methods used in the conversion of the qualitative decisions have been shown to give realistic results except for the decrease in the net income when the price-quality qualitative decisions moves towards reactive. That situation is shown to result from insufficient increase in the sale amount compared to the change in price. Such an unrealistic change of the net income can be avoided by updating the impact of the price score when the external factors are determined for a market behaviour. According to the current effect of the scores in the external factors, the market does not react sufficiently to a change in price. This causes a loss for the agents that expect earnings.

As a result of the experiments discussed in this chapter, the opposing agents have been shown to be capable of adapting to the changing environment, for instance by making expansion decisions when the capacity of their plant is not enough to satisfy the market demand and of making reasonable decisions autonomously.

# 7. RELATED WORK

This chapter reviews applications of Negoplan as well as selected applications of expert system shells similar to Negoplan.

## 7.1. RELATED WORK ON NEGOPLAN

Negoplan is a specialised system for representing decision problems. Planning, negotiations, distributed decision making and simulation are the examples of problem classes that have been implemented in Negoplan, so far.

The robot action planning application (Kersten *et al.*, 1994a) of Negoplan is an example of the planning problem class. In this application, the objectives of a robot navigating on an unknown terrain in order to accomplish a certain task are represented with a problem decomposition graph (PDG) and its actions with metarules. This application is used to observe the actions of the robot on an unknown terrain.

The international trade negotiations application (Kersten *et al.*, 1990) is an example of the negotiation problem class. This is an illustration of a contract negotiation case between two large companies. As a result of this application, it is expected to help one party to realise that negotiations are being carried on against their assumptions and expectations.

The resource allocation application (Kersten and Szpakowicz, 1994a) is an example of the distributed decision making problem class. In this application a limited resource/capacity situation is modelled where autonomous agents negotiate over resources to perform individual tasks.

The largest application of the Negoplan is Immelda (Kersten *et al.*, 1994b), a patient simulator presented in Section 3.1. This application is an example of the simulation class of problems.

The BSG application of Negoplan, presented in Chapter 5 and implemented using the extensions of Negoplan presented in Chapter 4, can also be considered as an example of the simulation class of problems. In this application a real-time environment is simulated for training purposes as in Immelda.

One of the most recent application of Negoplan before the BSG application is (Walls, 1997).

The extensions presented in Chapter 4 have been used recently in (Kersten and Szpakowicz, 1998). This work confirms the usability of the Negoplan extensions.

## 7.2. SIMILAR SYSTEMS AND THEIR APPLICATIONS

The Negoplan system is a domain-independent decision support system which can be customised for different application domains and it has the main characteristics of an expert system shell. Therefore, in this section, we concentrate on expert system shells with characteristics similar to Negoplan's that have been used to develop training tools.

There are several systems similar to Negoplan in terms of being domain-independent rule-based expert system shells. These systems are the Automated Reasoning Tool for Information Management (ART-IM), the C Language Integrated Production System (CLIPS), the Knowledge Engineering System (KES), LEVEL 5, and VAX OPS5. These systems are analysed in (Mettrey, 1991) for their functionalities, performance, advantages and disadvantages. Among these systems, CLIPS and VAX OPS5 have training tool applications that are similar to Negoplan's applications. Therefore, these systems are investigated in more detail in the rest of this section.

### 7.2.1. CLIPS

CLIPS (Giarratano and Riley, 1994) is a development and delivery expert system tool which provides a complete environment for the construction of rule and/or object based expert systems. It was designed to overcome a number of difficulties NASA had experienced using Lisp-based tools, including low availability of Lisp on conventional computers, high cost of Lisp-based tools and hardware and poor integration of Lisp with other languages.

CLIPS, by using rules as its primary knowledge representation approach and by having a forward-chaining inference engine, is very similar to the Negoplan system. It has a Lisp-like syntax for rules. Since CLIPS is implemented in standard C, it can be ported to all the hardware platforms that support a standard C compiler.

Originally CLIPS did not provide multi-agent capabilities. DYNACLIPS (Cengeloglu *et al.*, 1994) which is an implementation using CLIPS shells that runs on SunOS, provides a framework for dynamic knowledge exchange among intelligent agents. Intelligent agents can exchange facts, rules, and CLIPS commands at run time. This implementation adds similar capabilities and more to CLIPS as the extensions presented in Chapter 4 adds to Negoplan.

The Computer Aided Aircraft-design Package (CAAP) uses CLIPS to create an environment to design an airplane in an intelligent way for the engineering professionals and for students (Yalif, 1994). The rules defined using CLIPS in the system analyse the

design and if it does not satisfy the user-defined specifications, CAAP modifies the design considering the specification.

Some other applications of CLIPS that are used as training tools are (Germain and Desrosiers, 1991), (Kosta and Krolak, 1993), (Loftin *et al.*, 1989) and (Loftin *et al.*, 1987).

### 7.2.2. DEC OPS5

DEC OPS5 (Bronston *et al.*, 1985; Cooper and Wogrin, 1988) has advanced functionality over VAX OPS5 and is a language (Compiler and Run-Time Library) for constructing high-performance, forward chaining, rule-based applications. DEC OPS5 is a tool for developing rule-based systems. Systems developed with DEC OPS5 are well suited to solve problems in configuration, selection, diagnosis, process monitoring and control, scheduling, planning, decision support and rapid prototyping.

An application of DEC OPS5 is an intelligent system for tutoring phonetic transcription in introductory linguistics courses (Neubauer, 1992). This system guides the students through several attempts at transcribing a word with increasingly specific feedback, and an instructor can add, modify or delete data at any time with no assistance required from a programmer, as in medical decision-making application (Kersten *et al.*, 1994b) of Negoplan.

# 8. FUTURE WORK AND CONCLUSIONS

## 8.1. FUTURE WORK

In an application of the extensions presented in chapter 4, the Negoplan system has been used in the implementation of the managerial decision-making problem simulation based on the BSG software. However, the Negoplan system and the BSG application can both be improved. Certain possible improvements are discussed in the following two sections.

### 8.1.1. FUTURE WORK ON NEGOPLAN

The extensions of the Negoplan system enabled the implementation of the decision-making problem simulation with more than one opposing agent. Although the use of group references extended the expressiveness of the metarules and enabled the description of the common behaviour of a group of agents with generalised metarules, metarules still lack the power of expressing some behaviour more effectively.

In a decision-making problem with more than one opposing agent, some situations may occur as a result of an agreement between the members of a group. For example, consider the seller-buyers decision-making problem introduced in chapter 4. While the seller wants to sell its products for $100 each, if all buyers offer only $80, the seller may want to decrease the price asked in order to attract the buyers. This behaviour can be described by the following fictitious metarule:

```
buyer1 : 'gives $./item to buy . items'(80, _) ::= true &
buyer2 : 'gives $./item to buy . items'(80, _) ::= true &
buyer3 : 'gives $./item to buy . items'(80, _) ::= true &
buyer4 : 'gives $./item to buy . items'(80, _) ::= true &
buyer5 : 'gives $./item to buy . items'(80, _) ::= true &
seller : 'wants to sell ... pairs for $.../pair'(Amount,100) ::= true
==>
seller : 'wants to sell ... pairs for $.../pair'(Amount,90) ::= true.
```

Although the expected behaviour can be obtained by listing the metafacts for all buyers in the left-hand side of the metarule, this would require changing the metarule whenever the number of members in a group changes. In order to avoid this difficulty in maintaining an application, group names could be used in the left-hand side as the side names of metafacts required to exist for each of the group members for the metarule to be triggered. Since `buyer1` and `buyer2` are members of the `buyers1` group and `buyer3`, `buyer4` and

buyer5 are group members of the buyers2 group, the same behaviour could be described as follows:

```
buyers1 : 'gives $./item to buy . items'(80, _) ::= true &
buyers2 : 'gives $./item to buy . items'(80, _) ::= true &
seller : 'wants to sell ... pairs for $.../pair'(Amount,100) ::= true
==>
seller : 'wants to sell ... pairs for $.../pair'(Amount,90) ::= true.
```

This metarule will be triggered only if all members of the buyers1 and buyers2 groups satisfy the metafacts ascribed to their group names.

The same maintenance problem occurs in metarules that describe a conclusion which is true for all members of a group. For instance, in the seller-buyers decision-making problem, the buyers belonging to a group may decide to withdraw their offers to buy products, if the price goes above a certain limit. There is a need to repeat the metafact describing this reaction once for each group member in the right-hand-side of the metarule describing this behaviour:

```
seller : 'wants to sell ... pairs for $.../pair'(_,Price) ::= true &
{ Price > 150 }
==>
buyer1 : 'gives $./item to buy . items'(_, _) ::= false &
buyer2 : 'gives $./item to buy . items'(_, _) ::= false.
```

When this metarule is triggered the metafacts ascribed to the buyers1 group members are integrated with the existing metafacts. Since the use of the actual member names in a metarule creates the maintenance problem, the group name could again be used in the right-hand side of the metarules, but with an interpretation different than when it is used in the left-hand side. When a group name appears in the right-hand side, the metafact is added to the existing metafacts once for each member of the group. As a result of this interpretation, the metarule describing the reaction of a member of the buyers1 group could be described like this:

```
seller : 'wants to sell ... pairs for $.../pair'(_,Price) ::= true &
{ Price > 150 }
==>
buyers1 : 'gives $./item to buy . items'(_, _) ::= false.
```

These features have not yet been implemented because they were not crucial in the BSG application which requires essentially the ability to define more than one opposing agents.

The extensions have been implemented in the Macintosh version of Negoplan because it offers a more user-friendly environment essential for the user of the BSG application. The extensions will also be implemented on the Unix version of the Negoplan in the future.

### 8.1.2. Future work on the BSG application

The Business Strategy Game application of Negoplan could not be accomplished by respecting all the variety of the BSG software, because of the present performance of the Negoplan system on Macintosh computers. The number of geographical markets and the number of plants a company can operate are reduced to one, so that the BSG application could be implemented in Negoplan. However, if the application had been implemented on the Negoplan system on Unix systems, which has a better performance, all this variety could have been respected. After the Unix version is extended in order to allow the definition of more than one opposing agent, the BSG application can be re-implemented in full.

In the BSG application, the market is assumed to be equally sensitive to price, quality, service and advertisement criteria of the products. However, in reality some of those criteria may be more effective than others. For instance, while a certain percentage of the market demand may be for high quality and better serviced products, the rest may be for cheap and well advertised products. In the application, the tendencies that appear in the preferences of the individuals are not considered. This real-life behaviour could be achieved by associating each of those criteria with a factor between 0 and 1 denoting its relative importance. If the total of the factors for the four criteria is 1, the strategy score used in the determination of the market share of the companies could be computed as a weighted sum of criteria scores. Now, assume that the price, quality, service and advertisement factors are defined to be 0.3, 0.3, 0.2 and 0.2, respectively, suggesting that the price and the quality criteria are more effective in the final decision of the customer. The modification required in the determination of the companies' market shares to consider the different impacts of the four criteria could be achieved very easily by this method. However, in order to make the application simulate a particular geographical market, market research should be done in that market in order to determine the factors representing its behaviour.

Another aspect of real life business situations not considered in the BSG application is that the number of companies is fixed throughout the execution of the application. Currently, it is not possible to modify at run time the number of opposing agents participating in a decision-making problem. However, this extension could possibly be

integrated in the system and the behaviour describing the reactions to the modification of the number of opposing agents could also be simulated. One of the possible reactions to the modification in the number of companies would be the change in the average market demand per company, because the market demand does not depend on the number of companies.

The BSG application is implemented assuming that the market demand for products always grows randomly within a certain percentage. As a result of this assumption, there is always an increasing tendency in the price of the products, because the agents are assumed to spend more to get a higher market share. The increasing market demand can be modified to fluctuate and the behaviour of the agents can be extended so that they react to the negative changes in the market demand as well.

The BSG application has been tested in order to see if the opposing agents adapt their strategies to the changing environment and to their states. Therefore, it is only possible to claim that the BSG application is functionally useful. In order to claim that it also achieved its purpose in the practical education of the Business Administration students, it needs to be tested by a selected group of students. As a result of these tests, it may be necessary to make some modifications in the BSG application.

As a result of all these changes, the BSG application could approach closer to real life and could be more effective in the practical education of its users. However, this claim requires more experiments to show that the behaviour of the opposing agents is rational and the game is useful for business students.

The BSG application, as it is now, has been tested in order to see if the opposing agents adapt their strategies to the changing environment and to their own states; in other words, if they behave rationally. As a result of several executions of the BSG application, with a user who is constantly making the same decisions (always conservative or always aggressive), the opposing agents have been shown to react rationally to the changing environment. However, since these experiments have been performed by fixing the user's decisions, it may be unjustified to claim that the opposing agents behave rationally in all kinds of situations. Tests with many more executions are necessary. Since making decisions for the user manually increases substantially the time each execution takes and makes testing very time-consuming, there is a need to automate the user's decision-making process.

As a result of the tests that show the rationality of the opposing agents, it is only possible to claim that the BSG application is functionally useful. However, in order to claim that

the application also achieved its purpose in the practical education of the Business Administration students, it is still necessary to test the application with actual subjects.

The following two subsections present the plans that can be followed in the automation of the experiments and in experimenting with real-life subjects.

#### 8.1.2.1. Simulating the user's behaviour

The requirement of a large number of subjects and excessive amount of time to produce significant results are some of the reasons why automated agents can be very useful in testing interactive tools. An automated browsing agent, simulating the complex behavioural characteristics of a human searcher, has been used to test the effectiveness of an improvement in an interactive tool in (Drummond *et al.*, 1993; Holte and Drummond, 1994).

An automated agent can also be used to test the BSG application to see if the opposing agents behave rationally. The decision-making process of the supported agent whose control is given to a user can be remodelled to behave autonomously to simulate the behaviour of an actual student. This would substantially decrease the time necessary to execute the application. As a result, each execution can be run for as many decision-making periods as necessary and also the application can be run many times.

The simulated agent can behave rationally or irrationally. Irrational behaviour is necessary for the foolproof testing of the system to see the behaviour of the opposing agents in unexpected situations.

The test results can be evaluated by looking in detail at what decisions are made in what situations. For instance, the following questions can be asked to see whether the opposing agents behave rationally:

- what were the reactions of the opposing agents when they were not successful in one of the decision categories?
- in what conditions did the opposing agents make capacity increase decisions and what were their results?
- in what conditions did the opposing agents make plant automation decisions and what were their results?
- what happened when one of the agents made unexpectedly extreme decisions in one of the decision categories? how were the other agents affected by those decisions?

As a result of the answers collected for these kinds of questions, the rationality of the opposing agents can be determined, and some corrections may be made in order to improve the rational behaviour of the opposing agents.

#### 8.1.2.2. Testing of the BSG application by actual students

People who regularly face managerial decision-making problems can be considered the real target of the BSG application. Therefore, it may be the best to have the BSG application tested by the people who are currently experiencing managerial decision-making problems. However, this may not be very easy to achieve. First, it may not be easy to find enough people who are specialised in the same field as the BSG application (managing a footwear manufacturing company). It may also not be easy to observe the benefits those people acquired by using the application, because in real-life the results of managerial decisions can be observed only after several years. As a result of this, the BSG application should be tested by people on whom the results can be observed in shorter period of time. Those people can be the MBA students who are getting ready to play the same game using the original BSG software.

The game is played by MBA students in teams. Before the game begins, one of the teams can be selected and can be given a chance to practice by using the BSG application. As a result of practicing, that team might be expected to learn what decisions would give better results and to be more successful against the other teams in the actual game using the BSG software. The BSG application aims to be a training tool; therefore, the results obtained by the team practicing against the simulated opposing teams do not need to be considered when evaluating the usefulness of the BSG application. That is because those results alone may not necessarily mean that the BSG application has been successfully effective in the training of that team. This can be claimed to a certain extent only if that team succeeds, too, in the game against the actual teams. This can not be claimed to a full extent, because the BSG software may not reflect what exactly would happen in real life, either.

Apart from showing that the BSG application is a valuable training tool, tests that are performed using real-life subjects can also be important in the enhancement of the application. After practicing on the BSG application, the subjects can be given questionnaires to evaluate the application and according to the results of the evaluation the application can be enhanced.

## 8.2. CONCLUSIONS

The Negoplan system was designed and implemented to address decision-making problems that involve two participants and their environments. The restriction to two-party problems, which appear to be the majority of decision-making problems encountered in reality, reduces the usability of the Negoplan system.

The extensions presented in Chapter 4 have been designed and implemented to overcome this weakness of the Negoplan system. The extensions not only allowed the representation of more than two participants in a Negoplan application, but also introduced new constructs to augment the expressiveness of the Negoplan language. These constructs can describe the participants' similar behaviour and help achieve a substantial reduction in the amount of code required to implement new types of problem applications. As a result of these extensions, the problem domain addressed by the Negoplan system has been substantially expanded. A potential inefficiency in the system due to the increased number of participants in the broader class of decision-making problems has been minimised by optimisations in Negoplan's forward-chaining engine.

The extensions of the Negoplan system have been tested by two different applications, for two different purposes. The purpose of the first application was only to test the functionality of the extensions. The second application has been selected to demonstrate the usability of the extensions. As a result of this distinction, the latter ended up as an elaborate application of a real-life decision-making problem. This application has been based on a Business Strategy Game (BSG software) used in university education for training purposes.

The BSG software is used to simulate competition within a manufacturing industry with more than two participants. It was very suitable as a usability test. The usability of the extensions would not be validated, however, if the purpose were only to imitate the BSG software. Therefore, as well as imitating the main aspects of the BSG software, the BSG application has also been designed to overcome a weakness of the BSG software: the requirement of gathering several groups of students to play the game. Thus, the extensions have been used to get rid of this requirement to allow a student to play against participants represented by the system.

The application, presented in Section 4.4 and the BSG application presented in Chapter 5, have both been implemented; the extensions of the Negoplan system have been shown to give the expected results. As a result of the BSG application, a simulation of a manufacturing industry with autonomously behaving management teams has been

developed. The behaviour of the computer simulated management teams and the effects of this behaviour are discussed in Chapter 6.

This thesis presented the semantics and the implementation of the Negoplan system's extensions and the BSG application has demonstrated the usefulness of these extensions.

## 9. REFERENCES

L. Bronston, R. Farrell, E. Kant, N. Martin, *Programming expert systems in OPS5 : an introduction to rule-based programming*, Reading, Mass. ; Don Mills, Ont. : Addison-Wesley, 1985.

Y. Cengeloglu, S. Khajenoori and D. Linton, "DYNACLIPS (DYNAmic CLIPS): A Dynamic Knowledge Exchange Tool for Intelligent Agents", Third CLIPS Conference at NASA Johnson Space Center, Houston, TX, USA, pp. 353-366, September 1994.

T. Cooper and N. Wogrin, *Rule-based programming with OPS5*, San Mateo, Calif. : Morgan Kaufmann Publishers, 1988.

C. Drummond, R. Holte and D. Ionescu, "Accelerating Browsing by Automatically Inferring a User's Search Goal", *Proceedings of the Eighth Knowledge-Based Software Engineering Conference*, pp. 160-167, 1993.

A. Eliëns, *Principles of Object-Oriented Software Development*, Addison-Wesley Publishing Company, 1995.

D. Germain and S. Desrosiers, "Turning Up the Heat on Space Station Training: The Active Thermal Control System ICAT", *Proceedings of Contributed Sessions 1991 Conference on Intelligent Computer Aided Training*, Houston, TX, November 1991.

J. Giarratano and G. Riley, *Expert Systems: Principles and Programming*, Boston: PWS Pub. Co., 1994.

R. C. Holte and C. Drummond, "A Learning Apprentice For Browsing", AAAI Spring Symposium on Software Agents, 1994.

G. E. Kersten, "Simulation and Analysis of Negotiation Processes: The Case of Softwood Lumber Negotiations", *Proceedings of the 28th Hawaii International Conference on Systems Sciences. Volume IV: Information Systems: Collaboration Technology, Organizational Systems and Technology*, J.F. Nunamaker and R.H. Sprague, Jr. (eds), Los Alamitos, CA: IEEE Computer Society Press, pp. 242-261, 1995.

G. E. Kersten, L. Badcock, M. Iglewski and G. R. Mallory, "Structuring and Simulating Negotiations: An Approach and an Example", *Theory and Decision*, Vol. 28, No. 3, pp. 243-273, 1990.

G. E. Kersten, W. Michalowski, S. Szpakowicz and Z. Koperczak, "Restructurable Representations of Negotiation", *Management Science*, Vol. 37, No. 10, pp. 1269-1290, 1991.

G. E. Kersten, P. Lu, and S. Szpakowicz, "Indicative and Action Planning for an Intelligent Agent", *Proceedings of the Canadian Artificial Intelligence Conference*, Palo Alto, CA: Morgan Kaufman, pp. 287-294, 1994a.

G. E. Kersten, S. Rubin and S. Szpakowicz, "Medical Decision Making in Negoplan", *Moving Towards Expert Systems Globally in the 21st Century. Proceedings of the Second World Congress on Expert Systems*, J. Liebovitz (ed.), Cambridge, MA: Macmillan, pp. 1130-1137, 1994b.

G. E. Kersten and S. Szpakowicz, "Rule-based Formalism and Preference Representation: An Extension of NEGOPLAN". *European J. of Operational Research*, 45, pp.309-323, 1990.

G. E. Kersten and S. Szpakowicz, "Negotiation in Distributed Artificial Intelligence: Drawing from Human Experience", *Proceedings of the 27th Hawaii International Conference on Systems Sciences. Volume IV: Information Systems: Collaboration Technology, Organizational Systems and Technology*, J. F. Nunamaker and R. H. Sprague, Jr. (eds), Jan. 4-7, 1994, Los Alamitos, CA: IEEE Computer Society Press, pp. 258-270, 1994a.

G. E. Kersten and S. Szpakowicz, "A Formal Account of Sequential Decision-Making in a Co-operative Setting", *Fundamenta Informaticae*, vol. 30, no. 3/4, pp. 269-282, 1994b.

G. E. Kersten and S. Szpakowicz, "Modelling Business Negotiations for Electronic Commerce". *VII International Symposium on Intelligent Systems*, Malbork, Poland, June 1998 (11 pages, to appear).

C. P. Kosta and P. D. Krolak, "An Artificial Reality Environment for Remote Factory Control and Monitoring," *Vision 21: Interdisciplinary Science and Engineering in the Era of Cyberspace*, NASA/Lewis Research Center, December 1993.

R. B. Loftin, L. Wang, P. Baffes, G. Hua, "An Intelligent Training System for Space Shuttle Flight Controllers," *Innovative Applications of Artificial Intelligence*, AAAI Press/The MIT Press, Menlo Park, H. Schoor and A. Rappaport (eds), pp. 15-24, 1989.

R. B. Loftin *et al.*, "An Intelligent Training System for Payload-Assist Module Deploys," *Proceedings of The First Annual Workshop on Space Operations Automation and Robotics (SOAR '87)*, Houston, TX, August 1987.

W. Mettrey, "A Comparative Evaluation of Expert System Tools", *Computer*, pp. 19-31, February 1991.

S. J. Noronha and S. Szpakowicz, "Negoplan: A System for Logic-Based Decision Modelling", *Advances in Artificial Intelligence, Proc 11th Bienial Conf of the CSCSI*, AI'96, Toronto, G. McCalla (ed.), May 1996.

S. A. Ross, R. W. Westerfield, B. D. Jordan, G. S. Roberts, *Fundamentals of Corporate Finance*, second Canadian edition, pages 586-588, Irwin, 1993.

S. Szpakowicz, G. E. Kersten, and Z. Koperczak, "Knowledge-based decision support, preferences, and financial planning", *Proceedings of the 6th Conference on Artificial Intelligence Applications*, Santa Barbara, CA, USA, pp. 222-228, 1990.

A. A. Thompson, G. J. Stappenbeck, *The Business Strategy Game, A Global Industry Simulation*, Third Edition, University of Alabama, Irwin, 1995.

A. Walls, *Charon: A Computer Health Application for the Recognition and treatment Of Noxious plants*, University of Auckland, Unpublished Master's thesis, 1997.

G. U. Yalif, "The Computer Aided Aircraft-design Package (CAAP)", *Third CLIPS Conference at NASA Johnson Space Center*, Houston, TX, USA, pp. 263-272, September 1994.

## 10. NEGOPLAN BIBLIOGRAPHY

The following papers have been used in the research phase, although not all of them have been quoted in the thesis.

G. E. Kersten, "Simulation and Analysis of Negotiation Processes: The Case of Softwood Lumber Negotiations". *Proceedings of the 28th Hawaii International Conference on Systems Sciences. Volume IV: Information Systems: Collaboration Technology, Organizational Systems and Technology*, J.F. Nunamaker and R.H. Sprague, Jr. (eds), Los Alamitos, CA: IEEE Computer Society Press, pp. 242-261, 1995.

G. E. Kersten, L. Badcock, M. Iglewski and G. R. Mallory, "Structuring and Simulating Negotiations: An Approach and an Example", *Theory and Decision*, Vol. 28, No. 3, pp. 243-273, 1990.

G. E. Kersten, P. Lu, and S. Szpakowicz, "Indicative and Action Planning for an Intelligent Agent". *Proceedings of the Canadian Artificial Intelligence Conference*, Palo Alto, CA: Morgan Kaufman, pp. 287-294, 1994a.

G. E. Kersten, S. MacDonald, S. Rubin and S. Szpakowicz, "Knowledge-based Simulation for Medical Education", *Modelling and Simulation: Proceedings of the IASTED International Conference*, M.H. Hamza (ed.), Anaheim, CA: IASTED, pp. 630-633, 1993.

G. E. Kersten and W. Michalowski, "A Cooperative Expert System for Negotiation With a Hostage-Taker". *International Journal of Expert Systems*, Vol. 2, No. 3/4, pp. 357-376, 1989.

G. E. Kersten and S. Szpakowicz, "Rule-based Formalism and Preference Representation: An Extension of NEGOPLAN". *European J. of Operational Research*, 45, pp.309-323, 1990.

G. E. Kersten and S. Szpakowicz, "Decision Making and Decision Aiding: Defining the Process, Its Representations, and Support", *Group Decision and Negotiation*, Vol. 3, No. 2, pp. 237-261, 1994.

G. E. Kersten and S. Szpakowicz, "A Formal Account of Sequential Decision-Making in a Co-operative Setting", *Fundamenta Informaticae*, vol. 30, no. 3/4, pp. 269-282, 1994.

G. E. Kersten and S. Szpakowicz, "Negotiation in Distributed Artificial Intelligence: Drawing from Human Experience", *Proceedings of the 27th Hawaii International Conference on Systems Sciences. Volume IV: Information Systems: Collaboration Technology, Organizational Systems and Technology*, J. F. Nunamaker and R. H. Sprague, Jr. (eds), Jan. 4-7, 1994, Los Alamitos, CA: IEEE Computer Society Press, pp. 258-270, 1994.

G. E. Kersten and S. Szpakowicz, "Forming Decision Making Skills with a Patient Simulator", *Control and Cybernetics*, vol. 24(3), pp. 301-326, 1995.

G. E. Kersten, S. Rubin and S. Szpakowicz, "Medical Decision Making in Negoplan", *Moving Towards Expert Systems Globally in the 21st Century. Proceedings of the Second World Congress on Expert Systems*, J. Liebovitz (ed.), Cambridge, MA: Macmillan, pp. 1130-1137, 1994.

G. E. Kersten, S. Szpakowicz and Z. Koperczak, "Modelling of Decision Making for Decision Processes in Dynamic Environments", *Computers and Mathematics with Applications*, Vol. 20, No. 9/10, pp. 29-43, 1990.

G. E. Kersten, W. Michalowski, S. Szpakowicz and Z. Koperczak, "Restructurable Representations of Negotiation", *Management Science*, Vol. 37, No. 10, pp.1269-1290, 1991.

Z. Koperczak, G. E. Kersten and S. Szpakowicz, "The Negotiation Metaphor and Decision Support for Financial Modelling", in R.H. Spraque, Jr. (ed.) *Emerging Technologies and Applications. Proceedings of the 23rd Hawaii International Conference on System Sciences*, Los Alamitos, CA: IEEE Computer Society Press, pp. 31-40, 1990.

Z. Koperczak, S. Matwin and S. Szpakowicz, "Modelling Negotiation Strategies with Two Interacting Expert Systems", *Control and Cybernetics*, vol. 21(1), pp. 105-130, 1992.

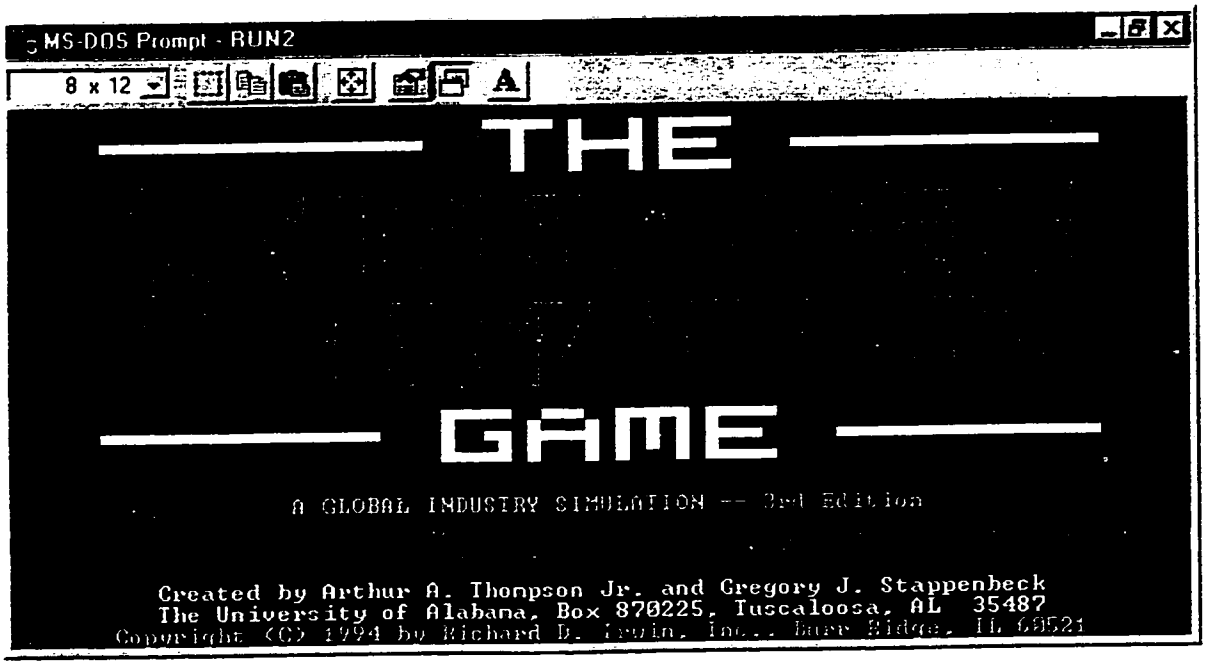
S. Matwin, S. Szpakowicz, Z. Koperczak, G. E. Kersten and W. Michalowski, "Negoplan: An Expert System Shell for Negotiation Support", *IEEE Expert*, Vol. 4, No. 4, pp. 50-62, 1989.

D. B. Meister and G. E. Kersten, "Restructurable Modelling in a Water Treatment Decision Support System", *Proceedings of the MCDM Conference*, Coimbra, Portugal, 1994.

S. Szpakowicz and G. E. Kersten, "Recent Developments in Negoplan", *Modelling and Simulation: Proceedings of the IASTED International Conference*, M.H. Hamza (ed.), Anaheim, CA: IASTED, pp. 126-129, 1993.

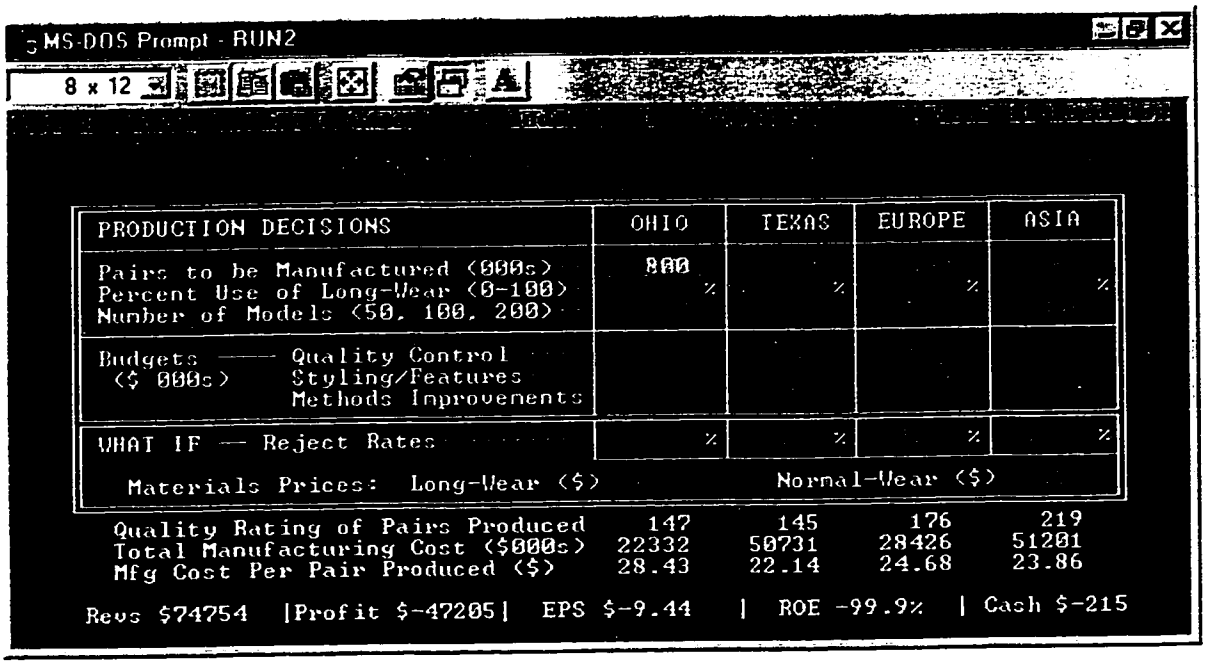
S. Szpakowicz, G. E. Kersten and Z. Koperczak, "Knowledge Based Decision Support, Preferences, and Financial Planning", *Proceedings of the 6th Conference on Artificial Intelligence Applications*, Piscataway, NJ: IEEE Press, pp. 222-228, 1990.

# APPENDIX A. SNAPSHOTS FROM THE BSG SOFTWARE




## Appendix A.1. THE BUSINESS STRATEGY GAME SOFTWARE

The BSG software is a DOS program and this is the initial screen of the software.




## Appendix A.2. THE PRODUCTION DECISIONS SCREEN

The students make their decisions by modifying the numerical values in the box. The numerical values that are outside the box are calculated, each time the student changes a value within the box. The values that are on the bottom of the screen are the financial indicators of the company. These values do not make sense because they reflect the state of the company only after all the decisions are made.

There are decision screens for each of the other decision groupings (Labour, Automation-Expansion, Marketing and Financial).

# APPENDIX B. THE METARULES OF THE BSG APPLICATION

## THE 'STUDENT_DECISION_MENU' PACKET

- The student makes a decision to switch to another packet.

```
student : makes_decision_year_round(_,_) ::= true
==>
select one ( make_high_level_decisions,
             make_low_level_decisions,
             end_decision_making_process,
             end_game )
with_message 'What is your choice ?'
--- student_decision_menu.
```

- The student selected `end_decision_making_process`, the qualitative and quantitative decisions stored in `stored_param2` clauses are extracted using the `read_param2` predicate and the `decisions` and `strategy` components of the PDG are virtually modified using `procedure` construct.

```
student : end_decision_making_process ::= true
==>
{ read_param2(student, production_decisions,
                    [ Quantity, Long_wear, Models, Quality_budget,
                    Styling_budget, Improvement_budget, Reject ] ),
  read_param2(student, labor_decisions,
                    [ Workers_employed, Annual_wage, Incentive_pay,
                    Worker_productivity ] ),
  read_param2(student, automation_expansion_decisions,
                    [ Automation_option, Expansion ] ),
  read_param2(student, market_decisions,
                    [ Wholesale_price, Advertising_budget, Dealers,
                    Rebates, Service_rep, Delivery_time, Sales_forecast]),
  read_param2(student, financial_decisions,
                    [ Short_term_loan, Shares_issued_retired, Bond_issue,
                    Dividend_declaration ] ),
  read_param2(student, short_term_strategy,
                    [ Price_strategy, Quality_Strategy, Service_Strategy,
                    Advertisement_Strategy ] )
} &
student : procedure (
      decisions <-
            production_decisions( Quantity, Long_wear, Models,
                  Quality_budget, Styling_budget, Improvement_budget,
                  Reject ) &
            labor_decisions( Workers_employed, Annual_wage,
                  Incentive_pay, Worker_productivity ) &
            automation_expansion_decisions(Automation_option,Expansion)&
            market_decisions( Wholesale_price, Advertising_budget,
                  Dealers, Rebates,Service_rep, Delivery_time,
                  Sales_forecast)&
            financial_decisions( Short_term_loan, Shares_issued_retired,
                  Bond_issue, Dividend_declaration ),
      strategy <-
            short_term_strategy(Price_strategy, Quality_Strategy,
                  Service_Strategy, Advertisement_Strategy) ) ::= true
--- student_decision_menu.
```

- The student selected `make_high_level_decisions`, the simulation switches to the `student_makes_high_level_decisions` packet. The `state_of_student` component of the PDG is modified.

```
student : make_high_level_decisions ::= true &
student : makes_decision_year_round(Year,Round) ::= true &
{ save_all_modifications(student) }
==>
modify ( state_of_student <-  makes_decision_year_round(Year,Round) &
                              makes_high_level_decisions(Year,1) )
switch_to 'student_makes_high_level_decisions'
--- student_decision_menu.
```

- The student selected `make_low_level_decisions`, the simulation switches to the `student_makes_low_level_decisions` packet. The `state_of_student` component of the PDG is modified.

```
student : make_low_level_decisions ::= true &
student : makes_decision_year_round(Year,Round) ::= true &
{ save_all_modifications(student) }
==>
modify ( state_of_student <- makes_decision_year_round(Year,Round) &
                             makes_low_level_decisions(Year,1) )
switch_to 'student_makes_low_level_decisions'
--- student_decision_menu.
```

- The student selected `end_decision_making_process`, the simulation switches to the `opposing_teams_define_strategy` packet. The `state_of_student` component of the PDG is modified.

```
student : end_decision_making_process ::= true &
student : makes_decision_year_round(Year,_) ::= true &
{ save_all_modifications(student) }
==>
modify ( state_of_student <- waits_for_opposing_teams_decisions(Year) )
switch_to 'opposing_teams_make_high_level_decisions'
--- student_decision_menu.
```

- The student selected `end_game`, the simulation terminates.

```
student : end_game ::= true
==>
terminate 'The game is terminated by the student'
--- student_decision_menu.
```

### THE 'STUDENT_MAKES_HIGH_LEVEL_DECISIONS' PACKET

- If the value of Round is 1 ( it is 1 when the packet is activated ), the student continues directly with the qualitative decision-making process.

```
student : makes_high_level_decisions(_Year, Round) ::= true &
{ Round = 1 }
==>
student : continue_high_level_decisions ::= true &
student : end_high_level_decisions ::= false
--- student_makes_high_level_decisions.
```

- If the value of Round is not 1 ( the value of Round is incremented after each cycle of the forward-chaining engine ), the student can either continue one more cycle in the student_makes_high_level_decisions packet or leave the packet.

```
student : makes_high_level_decisions(_Year, Round) ::= true &
{ Round > 1 }
==>
select one ( continue_high_level_decisions,
             end_high_level_decisions )
with_message 'Make a selection...'
--- student_makes_high_level_decisions.
```

- If the student continues the student_makes_high_level_decisions packet, the simulation proceeds with the short-term strategy definition of the student for the price, quality, service and advertisement decision categories.

```
student : continue_high_level_decisions ::= true &
{ read_param2(student, short_term_strategy,
                  [Old_Price_Strategy, Old_Quality_Strategy,
                  Old_Service_Strategy, Old_Advertisement_Strategy ] ) }
==>
select ( price(_Price_Strategy) +
            ask_sel( Old_Price_Strategy,
                        ['conservative', 'aggressive', 'reactive'] ),
         quality(_Quality_Strategy) +
            ask_sel( Old_Quality_Strategy,
                        ['conservative', 'aggressive', 'reactive'] ),
         service(_Service_Strategy) +
            ask_sel( Old_Service_Strategy,
                        ['conservative', 'aggressive', 'reactive'] ),
         advertisement(_Advertisement_Strategy) +
            ask_sel(Old_Advertisement_Strategy,
                        ['conservative', 'aggressive', 'reactive'] ) )
with_message 'Define your short-term strategy...'
--- student_makes_high_level_decisions.
```

- The student selected the price strategy ( this means the price strategy has been updated), then the new strategy is replaced in the hidden database and the new strategy is converted to quantitative decisions for Rebates and Improvement_budget. The Price decision is not yet made—this comes after all the other decisions.

```
student : price(Price_Strategy) ::= true &
{ read_param2(student, short_term_strategy,
            [_Old_Price_Strategy, Quality_Strategy,
            Service_Strategy, Advertisement_Strategy ] ) }
==>
{ write_param2(student, short_term_strategy,
```

```
            [Price_Strategy, Quality_Strategy,
            Service_Strategy, Advertisement_Strategy ] ),
  read_param2( student, market_decisions,
                        [Price, Adv_budget, Dealers, _Old_Rebates,
                        Service_rep, Delivery_time, Sales_forecast] ),
  read_param2( average, market_decisions,
                        [_, _, _, Avg_Rebates, _, _, _] ),
  read_param2( student, production_decisions,
                        [Quantity, Long_wear, Models, Quality_budget,
                        Styling_budget,_Old_Improvement_budget,Reject]),
  read_param2( average, production_decisions,
                        [_, _, _, _, _, Avg_Improvement_budget, _ ] ),
  new_rebates_wrt_strategy(Avg_Rebates, Price_Strategy, New_Rebates),
  strategy_coefficient( price, Price_Strategy, Coeff ),
  New_Improvement_budget is integer(Avg_Improvement_budget * Coeff),
  write_param2( student, production_decisions,
                        [Quantity, Long_wear, Models, Quality_budget,
                        Styling_budget,New_Improvement_budget, Reject]),
  write_param2( student, market_decisions,
                        [ Price, Adv_budget, Dealers, New_Rebates,
                        Service_rep, Delivery_time, Sales_forecast ])} &
student : price_strategy_converted ::= true
--- student_makes_high_level_decisions.
```

- The student did not select the `price` strategy and the current strategy is already converted.

```
student : price(_) ::= false
==>
student : price_strategy_converted ::= true
--- student_makes_high_level_decisions.
```

- The student selected `quality` strategy ( this means the `quality` strategy has been updated), then the new strategy is replaced in the hidden database and the new strategy is converted to quantitative decisions for `long_wear`, `models`, `quality_budget`, `styling_budget`, `annual_wage` and `incentive_pay`.

```
student : quality(Quality_Strategy) ::= true &
{ read_param2(student, short_term_strategy,
                        [Price_Strategy, _Old_Quality_Strategy,
                        Service_Strategy, Advertisement_Strategy ] ) }
==>
{ write_param2(student, short_term_strategy,
                        [Price_Strategy, Quality_Strategy,
                        Service_Strategy, Advertisement_Strategy ] ),
  read_param2( student, production_decisions,
                        [Quantity, _long_wear, _models, _quality_budget,
                        _styling_budget, Improvement_budget, Reject ] ),
  read_param2( average, production_decisions,
                        [_, Avg_long_wear, Avg_Models,
                        Avg_quality_budget, Avg_styling_budget, _, _ ]),
  strategy_coefficient( quality, Quality_Strategy, Coeff ),
  New_long_wear is integer(Avg_long_wear * Coeff),
  new_models_wrt_strategy( Avg_Models, Quality_Strategy, New_models ),
  New_quality_budget is integer(Avg_quality_budget * Coeff),
  New_styling_budget is integer(Avg_styling_budget * Coeff),
  write_param2(student, production_decisions,
                        [ Quantity, New_long_wear, New_models,
                        New_quality_budget, New_styling_budget,
                        Improvement_budget, Reject ] ),
```

```
  read_param2(student, labor_decisions,
                       [ Workers_employed, _annual_wage,
                       _incentive_pay, Worker_productivity ] ),
  read_param2(average, labor_decisions,
                       [ _, Avg_annual_wage, Avg_incentive_pay, _ ] ),
  New_annual_wage is integer(Avg_annual_wage * Coeff * 10) / 10,
  New_incentive_pay is integer(Avg_incentive_pay * Coeff * 10) / 10,
  write_param2(student, labor_decisions,
                       [ Workers_employed, New_annual_wage,
                       New_incentive_pay, Worker_productivity ] ) } &
student : quality_strategy_converted ::= true
--- student_makes_high_level_decisions.
```

- The student did not select the quality strategy and the current strategy is already converted.

```
student : quality(_) ::= false
==>
student : quality_strategy_converted ::= true
--- student_makes_high_level_decisions.
```

- The student selected service strategy ( this means the service strategy has been updated), then the new strategy is replaced in the hidden database and the new strategy is converted to quantitative decisions for Dealers, Service_rep and Delivery.

```
student : service(Service_Strategy) ::= true &
{ read_param2(student, short_term_strategy,
                       [Price_Strategy, Quality_Strategy,
                       _Old_Service_Strategy, Advertisement_Strategy])}
==>
{ write_param2(student, short_term_strategy,
                       [Price_Strategy, Quality_Strategy,
                       Service_Strategy, Advertisement_Strategy ] ),
  read_param2( student, market_decisions,
                       [Price, Adv_budget, _Dealers, Rebates,
                       _Service_rep, _Delivery, Sales_forecast ] ),
  read_param2( average, market_decisions,
                       [_, _, Avg_Dealers, _, Avg_Service_rep,
                       Avg_Delivery, _ ] ),
  strategy_coefficient( service, Service_Strategy, Coeff ),
  Inv_Coeff is 1/Coeff,
  New_Dealers is integer(Avg_Dealers * Coeff),
  Avg_Dealers_per_serv_rep is integer(Avg_Dealers / Avg_Service_rep),
  New_Dealers_per_serv_rep is Avg_Dealers_per_serv_rep * Inv_Coeff,
  New_Service_rep is integer(New_Dealers / New_Dealers_per_serv_rep ),
  new_delivery_wrt_strategy(Avg_Delivery,Service_Strategy,New_Delivery),
  write_param2(student, market_decisions,
                       [ Price, Adv_budget, New_Dealers, Rebates,
                       New_Service_rep,New_Delivery,Sales_forecast])} &
student : service_strategy_converted ::= true
--- student_makes_high_level_decisions.
```

- The student did not select the service strategy and the current strategy is already converted.

```
student : service(_) ::= false
==>
student : service_strategy_converted ::= true
--- student_makes_high_level_decisions.
```

- The student selected advertisement strategy ( this means the advertisement strategy has been updated), then the new strategy is replaced in the hidden database and the new strategy is converted to quantitative decisions for Adv_budget.

```
student : advertisement(Adv_Strategy) ::= true &
{ read_param2(student, short_term_strategy,
                       [Price_Strategy, Quality_Strategy,
                       Service_Strategy, _Old_Adv_Strategy ] ) }
==>
{ write_param2(student, short_term_strategy,
                       [Price_Strategy, Quality_Strategy,
                       Service_Strategy, Adv_Strategy ] ),
  read_param2( student, market_decisions,
                       [Price, _Adv_budget, Dealers, Rebates,
                       Service_rep, Delivery_time, Sales_forecast] ),
  read_param2( average, market_decisions,
                       [_, Avg_Adv_budget, _, _, _, _, _] ),
  strategy_coefficient( advertisement, Adv_Strategy, Coeff ),
  New_Adv_budget is integer(Avg_Adv_budget * Coeff ),
  write_param2(student, market_decisions,
                       [ Price, New_Adv_budget, Dealers, Rebates,
                       Service_rep, Delivery_time, Sales_forecast ])} &
student : advertisement_strategy_converted ::= true
--- student_makes_high_level_decisions.
```

- The student did not select the advertisement strategy and the current strategy is already converted.

```
student : advertisement(_) ::= false
==>
student : advertisement_strategy_converted ::= true
--- student_makes_high_level_decisions.
```

- After all the qualitative decisions have been converted to quantitative decisions, estimations are updated.

```
student : quality_strategy_converted ::= true &
student : service_strategy_converted ::= true &
student : advertisement_strategy_converted ::= true &
student : price_strategy_converted ::= true
==>
{ update_estimations(student, decision_making) } &
student : estimations_updated ::= true
--- student_makes_high_level_decisions.
```

- After the estimations have been updated, the production Quantity decision is made considering the Reject rate and Pairs_sold estimations and the Capacity of the plant.

```
student : estimations_updated ::= true &
student : inventory_from_last_year( Inventory_from_last_year,
            _Mfg_cost, _Avg_models, _Avg_quality) ::= true
==>
{ read_param2( student, production_decisions,
                       [_Quantity, Long_wear, Models, Quality_budget,
                       Styling_budget, Improvement_budget, Reject ] ),
  read_param2( student, market_decisions,
                       [_Price, _Adv_budget, _Dealers, _Rebates,
                       _Service_rep, _Delivery_time,
                       Estimated_Pairs_sold] ),
```

```
  Quantity_without_inventory is
          integer(Estimated_Pairs_sold/(1-(Reject/100))),
  New_Quantity is Quantity_without_inventory - Inventory_from_last_year,
  production_capacity(student,Capacity),
  smallest( [ Capacity, New_Quantity ], Smallest ),
  write_param2( student, production_decisions,
                         [Smallest, Long_wear, Models, Quality_budget,
                         Styling_budget, Improvement_budget, Reject ] )}&
student : number_of_pairs_decided ::= true
--- student_makes_high_level_decisions.
```

- After the production quantity is determined, the Number of `Workers_employed` decision is made considering the `Worker_productivity` estimation and the production `Quantity` decision.

```
student : number_of_pairs_decided ::= true
==>
{ cal_workers_needed(student, Workers_needed),
  read_param2(student, labor_decisions,
                         [ _Workers_employed, Annual_wage, Incentive_pay,
                         Worker_productivity ] ),
  write_param2(student, labor_decisions,
                         [ Workers_needed, Annual_wage, Incentive_pay,
                         Worker_productivity ] ) } &
student : number_of_workers_decided ::= true
--- student_makes_high_level_decisions.
```

- After all the other decisions have been made, the `Price` is made considering the operational cost of the product (`Oper_cost`) and the Price `Strategy`.

```
student : number_of_workers_decided ::= true &
{ read_param2(student, short_term_strategy, [Strategy, _, _, _] ) }
==>
{ read_param2( student, market_decisions,
                         [_Price, Adv_budget, Dealers, Rebates,
                         Service_rep, Delivery_time, Sales_forecast] ),
  strategy_coefficient( price, Strategy, Coeff ),
  cal_oper_cost_per_produced(student, Oper_cost),
  New_Price is integer(Oper_cost * Coeff * 100)/100,
  write_param2( student, market_decisions,
                         [ New_Price, Adv_budget, Dealers, Rebates,
                         Service_rep, Delivery_time, Sales_forecast ] )}&
student : price_decided ::= true
--- student_makes_high_level_decisions.
```

- After the short-term strategies have been converted to their corresponding quantitative decisions, a long-term strategy is determined. `automation_option` decision can only be made once in the life-time of the game. This is the case when it has not yet been made.

```
student : price_decided ::= true &
student : automation_option(_Year, none) ::= true &
{ read_param2(student, automation_expansion_decisions,
                         [ Old_Option, Old_Expansion ] ),
  convert2string_all( [ Old_Option, Old_Expansion ],
                      [ Str_Old_Option, Str_Old_Expansion] ) }
==>
select ( automation(_New_Option) +
           ask_sel( Str_Old_Option, [none, a, b, c, d]),
         increase_production_capacity(_New_Expansion) +
```

```
            ask_sel(Str_Old_Expansion, ['0', '1000', '2000', '3000'] ) )
with_message 'Define your long-term strategy...' &
student : longterm_strategy_defined ::= true
--- student_makes_high_level_decisions.
```

- This is the case when the `automation_option` decision has already been made.

```
student : price_decided ::= true &
student : automation_option(_Year, Existing_option) ::= true &
{ Existing_option \= none,
  read_param2(student, automation_expansion_decisions,
                       [ _Option, Old_Expansion ] ),
  convert2string_all( [ Old_Expansion ],
                      [ Str_Old_Expansion ] ) }
==>
select ( increase_production_capacity(_Capacity_increase) +
            ask_sel(Str_Old_Expansion, ['0', '1000', '2000', '3000'] ) )
with_message 'Define your long-term strategy :' &
student : longterm_strategy_defined ::= true
--- student_makes_high_level_decisions.
```

- The student selected `automation` and the `New_Option` is replaced in the hidden database.

```
student : automation(New_Option) ::= true
==>
{ read_param2(student, automation_expansion_decisions,
                       [ _Old_Option, Expansion ] ),
  write_param2(student, automation_expansion_decisions,
                       [ New_Option, Expansion ] ) }
--- student_makes_high_level_decisions.
```

- The student selected `increase_production_capacity` and the `New_Expansion` capacity is replaced in the hidden database.

```
student : increase_production_capacity(New_Expansion) ::= true
==>
{ read_param2(student, automation_expansion_decisions,
                       [ Automation_option, _Old_Expansion ] ),
  write_param2(student, automation_expansion_decisions,
                       [ Automation_option, New_Expansion ] ) }
--- student_makes_high_level_decisions.
```

- After the long-term strategy decisions have been made, the `financial_decisions` are made.

```
student : longterm_strategy_defined ::= true &
{ read_param2(student, financial_decisions,
                       [ Old_Short_term_loan, Old_Shares_issued,
                       Old_Bond_issue, Old_Dividend_declaration ] ),
  convert2string_all( [ Old_Short_term_loan, Old_Shares_issued,
                       Old_Bond_issue, Old_Dividend_declaration ],
                      [ Str_Short_term_loan, Str_Shares_issued,
                       Str_Bond_issue, Str_Dividend_declaration ])}
==>
select ( short_term_loan(_Short_term_loan) +
                       ask_int( Str_Short_term_loan, 0, 100000 ),
         shares_issued(_Shares_issued) +
                       ask_int( Str_Shares_issued, 0, 2000 ),
         bond_issue(_Bond_issue) + ask_int( Str_Bond_issue, 0, 100000 ),
         dividend_declaration(_Dividend_declaration) +
                       ask_real( Str_Dividend_declaration, 0, 1 ) )
```

```
with_message 'Define your financial strategy...' &
student : financial_decisions ::= true
--- student_makes_high_level_decisions.
```

- The student selected `short_term_loan` and the `New_Short_term_loan` is replaced in the hidden database.

```
student : short_term_loan(New_Short_term_loan) ::= true
==>
{ read_param2(student, financial_decisions,
                        [ _Old_Short_term_loan, Shares_issued,
                        Bond_issue, Dividend_declaration ] ),
  write_param2(student, financial_decisions,
                        [ New_Short_term_loan, Shares_issued,
                        Bond_issue, Dividend_declaration ] ) }
--- student_makes_high_level_decisions.
```

- The student selected `shares_issued` and the number of `New_Shares_issued` is replaced in the hidden database.

```
student : shares_issued(New_Shares_issued) ::= true
==>
{ read_param2(student, financial_decisions,
                        [ Short_term_loan, _Old_Shares_issued,
                        Bond_issue, Dividend_declaration ] ),
  write_param2(student, financial_decisions,
                        [ Short_term_loan, New_Shares_issued,
                        Bond_issue, Dividend_declaration ] ) }
--- student_makes_high_level_decisions.
```

- The student selected `bond_issue` and the amount earned by `New_Bond_issue` is replaced in the hidden database.

```
student : bond_issue(New_Bond_issue) ::= true
==>
{ read_param2(student, financial_decisions,
                        [ Short_term_loan, Shares_issued,
                        _Old_Bond_issue, Dividend_declaration ] ),
  write_param2(student, financial_decisions,
                        [ Short_term_loan, Shares_issued,
                        New_Bond_issue, Dividend_declaration ] ) }
--- student_makes_high_level_decisions.
```

- The student selected `dividend_declaration` and the `New_Dividend_declaration` is replaced in the hidden database.

```
student : dividend_declaration(New_Dividend_declaration) ::= true
==>
{ read_param2(student, financial_decisions,
                        [ Short_term_loan, Shares_issued, Bond_issue,
                        _Old_Dividend_declaration ] ),
  write_param2(student, financial_decisions,
                        [ Short_term_loan, Shares_issued, Bond_issue,
                        New_Dividend_declaration ] ) }
--- student_makes_high_level_decisions.
```

- After the `financial_decisions` have been made (they are the last decisions made in the `student_makes_high_level_decisions` packet), the estimated financial indicators are displayed to the user.

```
student : financial_decisions ::= true
==>
{ display_all_screens }
--- student_makes_high_level_decisions.
```

- The student did not select `end_high_level_decisions`, this means that he wants to continue this packet one more forward-chaining cycle. The `state_of_student` component of the PDG is modified.

```
student : end_high_level_decisions ::= false &
student : makes_high_level_decisions(Year,High_Level_Round) ::= true &
student : makes_decision_year_round(Year,Decision_Round) ::= true &
{ New_High_Level_Round is High_Level_Round + 1 }
==>
modify ( state_of_student <-
            makes_decision_year_round(Year,Decision_Round) &
            makes_high_level_decisions(Year, New_High_Level_Round) )
--- student_makes_high_level_decisions.
```

- The student selected `end_high_level_decisions`, this means that he wants to switch to the `student_decision_menu` packet. The `state_of_student` component of the PDG is modified.

```
student : end_high_level_decisions ::= true &
student : makes_decision_year_round(Year,Decision_Round) ::= true &
{ New_Decision_Round is Decision_Round + 1 }
==>
modify ( state_of_student <-
            makes_decision_year_round(Year,New_Decision_Round) )
switch_to 'student_decision_menu'
--- student_makes_high_level_decisions.
```

### THE 'STUDENT_MAKES_LOW_LEVEL_DECISIONS' PACKET

- The student selects the decision categories that he wants to modify and determines whether he wants to continue one more cycle in the student_makes_low_level_decisions packet.

```
student : makes_low_level_decisions(_Year,_Round) ::= true
==>
select ( production_decisions,
         labor_decisions,
         automation_expansion_decisions,
         marketing_decisions,
         financial_decisions,
         end_low_level_decisions )
with_message 'Make your selections...'
--- student_makes_low_level_decisions.
```

- The student selected the production_decisions category.

```
student : production_decisions ::= true &
{ read_param2(student, production_decisions,
                        [ Old_quantity, Old_long_wear, Old_models,
                        Old_quality_budget, Old_styling_budget,
                        Old_improvement_budget, _reject ] ),
  convert2string_all( [ Old_quantity, Old_long_wear, Old_models,
                        Old_quality_budget, Old_styling_budget,
                        Old_improvement_budget ],
                      [ Str_quantity, Str_long_wear, Str_models,
                        Str_quality_budget, Str_styling_budget,
                        Str_improvement_budget ] ),
  production_capacity(student,Capacity),
  Overtime_capacity is integer(Capacity * 1.2) }
==>
select ( quantity_produced(_Quantity) +
                ask_int( Str_quantity, 0, Overtime_capacity ),
         long_wear_percent(_Long_wear) +
                ask_int( Str_long_wear, 0, 100 ),
         number_of_models(_Models) +
                ask_sel( Str_models, [50,100,200] ),
         quality_budget(_Quality_budget) +
                ask_int( Str_quality_budget, 0, 10000 ),
         styling_budget(_Styling_budget) +
                ask_int( Str_styling_budget, 0, 10000 ),
         improvement_budget(_Improvement_budget) +
                ask_int( Str_improvement_budget, 0, 10000 ) )
with_message 'Make your production decisions...' &
student : production_decisions_made ::= true
--- student_makes_low_level_decisions.
```

- The student did not select the production_decisions category.

```
student : production_decisions ::= false
==>
student : production_decisions_updated ::= true
--- student_makes_low_level_decisions.
```

- The student selected quantity_produced in production_decisions category. The modification in this decision is updated in the hidden database.

```
student : quantity_produced(New_Quantity) ::= true
==>
```

```
{ read_param2(student, production_decisions,
                        [ _Old_Quantity, Long_wear, Models,
                       Quality_budget,Styling_budget,
                       Improvement_budget, Reject ] ),
  write_param2(student, production_decisions,
                       [ New_Quantity, Long_wear, Models,
                       Quality_budget, Styling_budget,
                       Improvement_budget, Reject] ) }
--- student_makes_low_level_decisions.
```

- The student selected `long_wear_percent` in the `production_decisions` category. The modification in this decision is updated in the hidden database.

```
student : long_wear_percent(New_Long_wear) ::= true
==>
{ read_param2(student, production_decisions,
                        [ Quantity, _Old_Long_wear, Models,
                       Quality_budget, Styling_budget,
                       Improvement_budget, Reject ] ),
  write_param2(student, production_decisions,
                       [ Quantity, New_Long_wear, Models,
                       Quality_budget, Styling_budget,
                       Improvement_budget, Reject] ) }
--- student_makes_low_level_decisions.
```

- The student selected `number_of_models` in the `production_decisions` category. The modification in this decision is updated in the hidden database.

```
student : number_of_models(New_Models) ::= true
==>
{ read_param2(student, production_decisions,
                        [ Quantity, Long_wear, _Old_Models,
                       Quality_budget, Styling_budget,
                       Improvement_budget, Reject ] ),
  write_param2(student, production_decisions,
                       [ Quantity, Long_wear, New_Models,
                       Quality_budget, Styling_budget,
                       Improvement_budget, Reject] ) }
--- student_makes_low_level_decisions.
```

- The student selected `quality_budget` in the `production_decisions` category. The modification in this decision is updated in the hidden database.

```
student : quality_budget(New_Quality_budget) ::= true
==>
{ read_param2(student, production_decisions,
                        [ Quantity, Long_wear, Models,
                       _Old_Quality_budget, Styling_budget,
                       Improvement_budget, Reject ] ),
  write_param2(student, production_decisions,
                       [ Quantity, Long_wear, Models,
                       New_Quality_budget, Styling_budget,
                       Improvement_budget, Reject] ) }
--- student_makes_low_level_decisions.
```

- The student selected `styling_budget` in the `production_decisions` category. The modification in this decision is updated in the hidden database.

```
student : styling_budget(New_Styling_budget) ::= true
==>
{ read_param2(student, production_decisions,
```

```
                          [ Quantity, Long_wear, Models, Quality_budget,
                          _Old_Styling_budget,Improvement_budget,Reject]),
  write_param2(student, production_decisions,
                          [ Quantity, Long_wear, Models, Quality_budget,
                          New_Styling_budget, Improvement_budget,Reject])}
--- student_makes_low_level_decisions.
```

- The student selected `improvement_budget` in the `production_decisions` category. The modification in this decision is updated in the hidden database.

```
student : improvement_budget(New_Improvement_budget) ::= true
==>
{ read_param2(student, production_decisions,
                          [ Quantity, Long_wear, Models, Quality_budget,
                          Styling_budget,_Old_Improvement_budget,Reject]),
  write_param2(student, production_decisions,
                          [ Quantity, Long_wear, Models, Quality_budget,
                          Styling_budget, New_Improvement_budget,Reject])}
--- student_makes_low_level_decisions.
```

- If `production_decisions_made`, display the production screen.

```
student : production_decisions_made ::= true
==>
{ update_estimations(student,decision_making),
  display_production_screen } &
student : production_decisions_updated ::= true
--- student_makes_low_level_decisions.
```

- The student selected the `labor_decisions` category and the production decisions are updated.

```
student : labor_decisions ::= true &
student : production_decisions_updated ::= true &
{ read_param2(student, labor_decisions,
                          [ Old_workers_employed, Old_annual_wage,
                          Old_incentive_pay, _worker_productivity ] ),
  convert2string_all( [ Old_workers_employed, Old_annual_wage,
                          Old_incentive_pay ],
                        [ Str_workers_employed, Str_annual_wage,
                          Str_incentive_pay ] ),
  minimum_annual_wage(Minimum_annual_wage),
  cal_workers_needed(student, Workers_needed) }
==>
select ( workers_employed(_Workers_employed) +
                  ask_int( Str_workers_employed, 0, Workers_needed ),
         annual_wage(_Annual_wage) +
                  ask_real( Str_annual_wage, Minimum_annual_wage,100.0),
         incentive_pay(_Incentive_pay) +
                  ask_real( Str_incentive_pay, 0, 10 ) )
with_message 'Make your labor decisions...' &
student : labor_decisions_made ::= true
--- student_makes_low_level_decisions.
```

- The student did not select the `labor_decisions` category.

```
student : labor_decisions ::= false
==>
student : labor_decisions_updated ::= true
--- student_makes_low_level_decisions.
```

- The student selected workers_employed in labor_decisions category. The modification in this decision is updated in the hidden database.

```
student : workers_employed(New_Workers_employed) ::= true
==>
{ read_param2(student, labor_decisions,
                        [ _Old_Workers_employed, Annual_wage,
                        Incentive_pay, Worker_productivity ] ),
  write_param2(student, labor_decisions,
                        [ New_Workers_employed, Annual_wage,
                        Incentive_pay, Worker_productivity ] ) }
--- student_makes_low_level_decisions.
```

- The student selected annual_wage in the labor_decisions category. The modification in this decision is updated in the hidden database.

```
student : annual_wage(New_Annual_wage) ::= true
==>
{ read_param2(student, labor_decisions,
                        [ Workers_employed, _Old_Annual_wage,
                        Incentive_pay, Worker_productivity ] ),
  write_param2(student, labor_decisions,
                        [ Workers_employed, New_Annual_wage,
                        Incentive_pay, Worker_productivity ] ) }
--- student_makes_low_level_decisions.
```

- The student selected incentive_pay in the labor_decisions category. The modification in this decision is updated in the hidden database.

```
student : incentive_pay(New_Incentive_pay) ::= true
==>
{ read_param2(student, labor_decisions,
                        [ Workers_employed, Annual_wage,
                        _Old_Incentive_pay, Worker_productivity ] ),
  write_param2(student, labor_decisions,
                        [ Workers_employed, Annual_wage,
                        New_Incentive_pay, Worker_productivity ] ) }
--- student_makes_low_level_decisions.
```

- If labor_decisions_made, display the labor screen.

```
student : labor_decisions_made ::= true
==>
{ update_estimations(student,decision_making),
  display_labor_screen } &
student : labor_decisions_updated ::= true
--- student_makes_low_level_decisions.
```

- The student selected the automation_expansion_decisions category and the labor decisions are updated. The automation decision has not yet been made.

```
student : automation_expansion_decisions ::= true &
student : labor_decisions_updated ::= true &
student : automation_option(_Year, none) ::= true &
{ read_param2(student, automation_expansion_decisions,
                        [ Old_Automation_option, Old_Expansion ] ),
  convert2string_all( [ Old_Automation_option, Old_Expansion ],
                      [ Str_Automation_option, Str_Expansion ] ) }
==>
select ( automation(_Automation_option) +
                  ask_sel( Str_Automation_option, [ a, b, c, d, none ]),
```

```
        expansion_of_plant(_Expansion) +
                    ask_sel( Str_Expansion, ['0','1000','2000','3000'] ) )
with_message 'Make your automation expansion decisions...' &
student : automation_expansion_decisions_made ::= true
--- student_makes_low_level_decisions.
```

- The student selected the automation_expansion_decisions category and the labor decisions are updated. The automation decision has already been made, in a previous year.

```
student : automation_expansion_decisions ::= true &
student : labor_decisions_updated ::= true &
student : automation_option(_Year, Option) ::= true &
{ Option \= none,
  read_param2(student, automation_expansion_decisions,
                       [ _Automation_option, Old_Expansion ] ),
  convert2string_all( [ Old_Expansion ], [ Str_Expansion ] ),
  give_warning('This plant is already automated!', 500) }
==>
select ( expansion_of_plant(_Expansion) +
                    ask_sel( Str_Expansion, ['0','1000','2000','3000'] ) )
with_message 'Make your expansion decision...' &
student : automation_expansion_decisions_made ::= true
--- student_makes_low_level_decisions.
```

- The student did not select the automation_expansion_decisions category.

```
student : automation_expansion_decisions ::= false
==>
student : automation_expansion_decisions_updated ::= true
--- student_makes_low_level_decisions.
```

- The student selected automation in the automation_expansion_decisions category. The modification in this decision is updated in the hidden database.

```
student : automation(New_Automation_option) ::= true
==>
{ read_param2(student, automation_expansion_decisions,
                       [ _Old_Automation_option, Expansion ] ),
  write_param2(student, automation_expansion_decisions,
                       [ New_Automation_option, Expansion ] ) }
--- student_makes_low_level_decisions.
```

- The student selected expansion_of_plant in the automation_expansion_decisions category. The modification in this decision is updated in the hidden database.

```
student : expansion_of_plant(New_Expansion) ::= true
==>
{ read_param2(student, automation_expansion_decisions,
                       [Automation_option, _Old_Expansion ] ),
  write_param2(student, automation_expansion_decisions,
                       [Automation_option, New_Expansion ] ) }
--- student_makes_low_level_decisions.
```

- If automation_expansion_decisions_made, display the automation-expansion screen.

```
student : automation_expansion_decisions_made ::= true
==>
{ update_estimations(student,decision_making),
```

```
  display_automation_expansion_screen ) &
student : automation_expansion_decisions_updated ::= true
--- student_makes_low_level_decisions.
```

- The student selected the marketing_decisions category and the automation-expansion decisions are updated.

```
student : marketing_decisions ::= true &
student : automation_expansion_decisions_updated ::= true &
{ read_param2(student, market_decisions,
                        [ Old_Wholesale_price, Old_Advertising_budget,
                        Old_Dealers, Old_Rebates, Old_Service_rep,
                        Old_Delivery_time, _Sales_forecast ] ),
  convert2string_all( [ Old_Wholesale_price, Old_Advertising_budget,
                        Old_Dealers, Old_Rebates, Old_Service_rep,
                        Old_Delivery_time ],
                      [ Str_Wholesale_price, Str_Advertising_budget,
                        Str_Dealers, Str_Rebates, Str_Service_rep,
                        Str_Delivery_time ] ) }
==>
select ( wholesale_price(_Wholesale_price) +
                ask_real( Str_Wholesale_price, 0, 100 ),
         advertising_budget(_Advertising_budget) +
                ask_int( Str_Advertising_budget, 0, 10000 ),
         dealers(_Dealers) + ask_int( Str_Dealers, 0, 2000 ),
         rebates(_Rebates) + ask_sel( Str_Rebates, ['0', '1', '3','5']),
         service_rep(_Service_rep) + ask_int( Str_Service_rep, 0, 100 ),
         delivery_time(_Delivery_time) +
                ask_sel( Str_Delivery_time, [1, 2, 3, 4] ) )
with_message 'Make your marketing decisions...' &
student : marketing_decisions_made ::= true
--- student_makes_low_level_decisions.
```

- The student did not select the marketing_decisions category.

```
student : marketing_decisions ::= false
==>
student : marketing_decisions_updated ::= true
--- student_makes_low_level_decisions.
```

- The student selected wholesale_price in the marketing_decisions category. The modification in this decision is updated in the hidden database.

```
student : wholesale_price(New_Wholesale_price) ::= true
==>
{ read_param2(student, market_decisions,
                        [ _Old_Wholesale_price, Advertising_budget,
                        Dealers, Rebates, Service_rep, Delivery_time,
                        Sales_forecast ] ),
  write_param2(student, market_decisions,
                        [ New_Wholesale_price, Advertising_budget,
                        Dealers, Rebates, Service_rep, Delivery_time,
                        Sales_forecast ] ) }
--- student_makes_low_level_decisions.
```

- The student selected advertising_budget in the marketing_decisions category. The modification in this decision is updated in the hidden database.

```
student : advertising_budget(New_Advertising_budget) ::= true
==>
{ read_param2(student, market_decisions,
                        [ Wholesale_price, _Old_Advertising_budget,
```

```
                          Dealers, Rebates, Service_rep, Delivery_time,
                          Sales_forecast ] ),
  write_param2(student, market_decisions,
                          [ Wholesale_price, New_Advertising_budget,
                          Dealers, Rebates, Service_rep, Delivery_time,
                          Sales_forecast ] ) }
--- student_makes_low_level_decisions.
```

- The student selected `dealers` in the `marketing_decisions` category. The modification in this decision is updated in the hidden database.

```
student : dealers(New_Dealers) ::= true
==>
{ read_param2(student, market_decisions,
                          [ Wholesale_price, Advertising_budget,
                          _Old_Dealers, Rebates, Service_rep,
                          Delivery_time, Sales_forecast ] ),
  write_param2(student, market_decisions,
                          [ Wholesale_price, Advertising_budget,
                          New_Dealers, Rebates, Service_rep,
                          Delivery_time, Sales_forecast ] ) }
--- student_makes_low_level_decisions.
```

- The student selected `rebates` in the `marketing_decisions` category. The modification in this decision is updated in the hidden database.

```
student : rebates(New_Rebates) ::= true
==>
{ read_param2(student, market_decisions,
                          [ Wholesale_price, Advertising_budget, Dealers,
                          _Old_Rebates, Service_rep, Delivery_time,
                          Sales_forecast ] ),
  write_param2(student, market_decisions,
                          [ Wholesale_price, Advertising_budget, Dealers,
                          New_Rebates, Service_rep, Delivery_time,
                          Sales_forecast ] ) }
--- student_makes_low_level_decisions.
```

- The student selected `service_rep` in the `marketing_decisions` category. The modification in this decision is updated in the hidden database.

```
student : service_rep(New_Service_rep) ::= true
==>
{ read_param2(student, market_decisions,
                          [ Wholesale_price, Advertising_budget, Dealers,
                          Rebates, _Old_Service_rep, Delivery_time,
                          Sales_forecast ] ),
  write_param2(student, market_decisions,
                          [ Wholesale_price, Advertising_budget, Dealers,
                          Rebates, New_Service_rep, Delivery_time,
                          Sales_forecast ] ) }
--- student_makes_low_level_decisions.
```

- The student selected `delivery_time` in the `marketing_decisions` category. The modification in this decision is updated in the hidden database.

```
student : delivery_time(New_Delivery_time) ::= true
==>
{ read_param2(student, market_decisions,
                          [ Wholesale_price, Advertising_budget, Dealers,
                          Rebates, Service_rep, _Old_Delivery_time,
```

```
                          Sales_forecast ] ),
  write_param2(student, market_decisions,
                          [ Wholesale_price, Advertising_budget, Dealers,
                          Rebates, Service_rep, New_Delivery_time,
                          Sales_forecast ] ) }
--- student_makes_low_level_decisions.
```

- If `marketing_decisions_made`, display the marketing screen.

```
student : marketing_decisions_made ::= true
==>
{ update_estimations(student,decision_making),
  display_marketing_screen } &
student : marketing_decisions_updated ::= true
--- student_makes_low_level_decisions.
```

- The student selected the `financial_decisions` category and the marketing decisions are updated.

```
student : financial_decisions ::= true &
student : marketing_decisions_updated ::= true &
{ read_param2(student, financial_decisions,
                          [ Old_Short_term_loan, Old_Shares_issued,
                          Old_Bond_issue, Old_Dividend_declaration ] ),
  convert2string_all( [ Old_Short_term_loan, Old_Shares_issued,
                          Old_Bond_issue, Old_Dividend_declaration ],
                        [ Str_Short_term_loan, Str_Shares_issued,
                          Str_Bond_issue, Str_Dividend_declaration ] ) }
==>
select ( short_term_loan(_Short_term_loan) +
                  ask_int( Str_Short_term_loan, 0, 100000 ),
         shares_issued(_Shares_issued) +
                  ask_int( Str_Shares_issued, 0, 2000 ),
         bond_issue(_Bond_issue) + ask_int( Str_Bond_issue, 0, 100000 ),
         dividend_declaration(_Dividend_declaration) +
                  ask_real( Str_Dividend_declaration, 0, 1 ) )
with_message 'Make your financial decisions...' &
student : financial_decisions_made ::= true
--- student_makes_low_level_decisions.
```

- The student did not select the `financial_decisions` category.

```
student : financial_decisions ::= false
==>
student : financial_decisions_updated ::= true
--- student_makes_low_level_decisions.
```

- The student selected `short_term_loan` in the `financial_decisions` category. The modification in this decision is updated in the hidden database.

```
student : short_term_loan(New_Short_term_loan) ::= true
==>
{ read_param2(student, financial_decisions,
                          [ _Old_Short_term_loan, Shares_issued,
                          Bond_issue, Dividend_declaration ] ),
  write_param2(student, financial_decisions,
                          [ New_Short_term_loan, Shares_issued,
                          Bond_issue, Dividend_declaration ] )}
--- student_makes_low_level_decisions.
```

- The student selected `shares_issued` in the `financial_decisions` category. The modification in this decision is updated in the hidden database.

```
student : shares_issued(New_Shares_issued) ::= true
==>
{ read_param2(student, financial_decisions,
                        [ Short_term_loan, _Old_Shares_issued,
                        Bond_issue, Dividend_declaration ] ),
  write_param2(student, financial_decisions,
                        [ Short_term_loan, New_Shares_issued,
                        Bond_issue, Dividend_declaration ] )}
--- student_makes_low_level_decisions.
```

- The student selected `bond_issue` in the `financial_decisions` category. The modification in this decision is updated in the hidden database.

```
student : bond_issue(New_Bond_issue) ::= true
==>
{ read_param2(student, financial_decisions,
                        [ Short_term_loan, Shares_issued,
                        _Old_Bond_issue, Dividend_declaration ] ),
  write_param2(student, financial_decisions,
                        [ Short_term_loan, Shares_issued,
                        New_Bond_issue, Dividend_declaration ] )}
--- student_makes_low_level_decisions.
```

- The student selected `dividend_declaration` in the `financial_decisions` category. The modification in this decision is updated in the hidden database.

```
student : dividend_declaration(New_Dividend_declaration) ::= true
==>
{ read_param2(student, financial_decisions,
                        [ Short_term_loan, Shares_issued,
                        Bond_issue, _Old_Dividend_declaration ] ),
  write_param2(student, financial_decisions,
                        [ Short_term_loan, Shares_issued,
                        Bond_issue, New_Dividend_declaration ] )}
--- student_makes_low_level_decisions.
```

- If `financial_decisions_made`, display the financial screen.

```
student : financial_decisions_made ::= true
==>
{ update_estimations(student,decision_making),
  display_financial_screen } &
student : financial_decisions_updated ::= true
--- student_makes_low_level_decisions.
```

- The student did not select `end_low_level_decisions`, this means that he wants to continue this packet one more forward-chaining cycle. The `state_of_student` component of the PDG is modified.

```
student : end_low_level_decisions ::= false &
student : makes_low_level_decisions(Year,Low_Level_Round) ::= true &
student : makes_decision_year_round(Year,Decision_Round) ::= true &
{ New_Low_Level_Round is Low_Level_Round + 1 }
==>
modify ( state_of_student <-
            makes_decision_year_round(Year,Decision_Round) &
            makes_low_level_decisions(Year, New_Low_Level_Round) )
--- student_makes_low_level_decisions.
```

- The student selected `end_low_level_decisions`, this means that he wants to switch to the `student_decision_menu` packet. The `state_of_student` component of the PDG is modified.

```
student : end_low_level_decisions ::= true &
student : makes_decision_year_round(Year,Decision_Round) ::= true &
{ New_Decision_Round is Decision_Round + 1 }
==>
modify ( state_of_student <-
            makes_decision_year_round(Year,New_Decision_Round) )
switch_to 'student_decision_menu'
--- student_makes_low_level_decisions.
```

### THE 'OPPOSING_TEAMS_MAKE_HIGH_LEVEL_DECISIONS' PACKET

- Initialisation of the opposing teams' states and their `long_term_financing_amount` predicates.

```
student : waits_for_opposing_teams_decisions(_Year) ::= true
==>
opposing_team_1 : defines_strategy ::= true &
opposing_team_2 : defines_strategy ::= true &
opposing_team_1 : long_term_financing_amount(0) ::= true &
opposing_team_2 : long_term_financing_amount(0) ::= true
--- opposing_teams_make_high_level_decisions.
```

- If last year's `Price_score` is less than 1.0 ( average score ), the short-term price strategy is `reactive`. The new strategy is updated in the `short_term_strategy` metafact as well as in the hidden database.

```
opposing_team : defines_strategy ::= true  &
opposing_team : price_qual_serv_adv_scores(
                          Price_score, _, _, _) ::= true &
{ existing_metafact( opposing_team, short_term_strategy
            (_Old_Price, Quality, Service, Advertisement), true),
  score_threshold(price, Threshold),
  Score is Price_score + Threshold,
  Score < 1.0 }
==>
opposing_team : short_term_strategy
            ( reactive, Quality, Service, Advertisement ) ::= true &
opposing_team : price_strategy_defined ::= true &
{ write_param2(opposing_team, short_term_strategy,
            [reactive, Quality, Service, Advertisement ] ) }
--- opposing_teams_make_high_level_decisions.
```

- If last year's `Price_score` is greater than or equal to 1.0 ( average score ), the short-term price strategy is `conservative` or `aggressive`. The new strategy is updated in the `short_term_strategy` metafact as well as in the hidden database.

```
opposing_team : defines_strategy ::= true  &
opposing_team : price_qual_serv_adv_scores(
                          Price_score, _, _, _) ::= true &
{ existing_metafact( opposing_team, short_term_strategy
            (_Old_Price, Quality, Service, Advertisement), true),
  score_threshold(price, Threshold),
  Score is Price_score + Threshold,
  Score >= 1.0,
  random_member(New_Price_Stgy, [conservative, conservative]) }
==>
opposing_team : short_term_strategy
            ( New_Price_Stgy, Quality, Service, Advertisement ) ::=true&
opposing_team : price_strategy_defined ::= true &
{ write_param2(opposing_team, short_term_strategy,
            [New_Price_Stgy, Quality, Service, Advertisement ] ) }
--- opposing_teams_make_high_level_decisions.
```

- Conversion of the `price` strategy to quantitative decisions for `Rebates` and `Improvement_budget`. The `Price` decision will be made after all the other decisions.

```
opposing_team : price_strategy_defined ::= true &
{ read_param2(opposing_team, short_term_strategy,
            [Price_Strategy, _, _, _ ] ),
```

```
  read_param2(opposing_team, market_decisions,
              [Price, Adv_budget, Dealers, _Old_Rebates,
              Service_rep, Delivery_time, Sales_forecast] ),
  read_param2(average, market_decisions, [_, _, _, Avg_Rebates, _,_,_]),
  read_param2(opposing_team, production_decisions,
              [Quantity, Long_wear, Models, Quality_budget,
              Styling_budget, _Old_Improvement_budget, Reject ] ),
  read_param2(average, production_decisions,
              [ _, _, _, _, _, Avg_Improvement_budget, _] ),
  new_rebates_wrt_strategy(Avg_Rebates, Price_Strategy, New_Rebates),
  strategy_coefficient( price, Price_Strategy, Coeff ),
  New_Improvement_budget is integer(Avg_Improvement_budget * Coeff) }
==>
{ write_param2(opposing_team, production_decisions,
              [Quantity, Long_wear, Models, Quality_budget,
              Styling_budget, New_Improvement_budget, Reject ] ),
  write_param2(opposing_team, market_decisions,
              [ Price, Adv_budget, Dealers, New_Rebates,
              Service_rep, Delivery_time, Sales_forecast ] ) } &
opposing_team : price_strategy_defined ::= false &
opposing_team : price_strategy_converted ::= true
--- opposing_teams_make_high_level_decisions.
```

- If last year's `Quality_score` is less than 1.0 ( average score ), the short-term quality strategy is `reactive`. The new strategy is updated in the `short_term_strategy` metafact as well as in the hidden database.

```
opposing_team : defines_strategy ::= true &
opposing_team : price_qual_serv_adv_scores(
                          _, Quality_score, _, _) ::= true &
{ existing_metafact( opposing_team, short_term_strategy
              ( Price, _Old_Quality, Service, Advertisement), true),
  score_threshold(quality, Threshold),
  Score is Quality_score + Threshold,
  Score < 1.0 }
==>
opposing_team : short_term_strategy
              ( Price, reactive, Service, Advertisement ) ::= true &
opposing_team : quality_strategy_defined ::= true &
{ write_param2(opposing_team, short_term_strategy,
              [Price, reactive, Service, Advertisement ] ) }
--- opposing_teams_make_high_level_decisions.
```

- If last year's `Quality_score` is greater than or equal to 1.0 ( average score ), the short-term quality strategy is `conservative` or `aggressive`. The new strategy is updated in the `short_term_strategy` metafact as well as in the hidden database.

```
opposing_team : defines_strategy ::= true  &
opposing_team : price_qual_serv_adv_scores(
                          _, Quality_score, _, _) ::= true &
{ existing_metafact( opposing_team, short_term_strategy
              ( Price, _Old_Quality, Service, Advertisement), true),
  score_threshold(quality, Threshold),
  Score is Quality_score + Threshold,
  Score >= 1.0,
  random_member(New_Quality_Stgy, [conservative, aggressive]) }
==>
opposing_team : short_term_strategy
              ( Price, New_Quality_Stgy, Service, Advertisement ) ::=true&
opposing_team : quality_strategy_defined ::= true &
```

```
{ write_param2(opposing_team, short_term_strategy,
            [Price, New_Quality_Stgy, Service, Advertisement ] ) }
--- opposing_teams_make_high_level_decisions.
```

- Conversion of the `quality` strategy to quantitative decisions for `long_wear`, `models`, `quality_budget`, `styling_budget`, `annual_wage` and `incentive_pay`.

```
opposing_team : quality_strategy_defined ::= true &
{ read_param2(opposing_team, short_term_strategy,
            [_, Quality_Strategy, _, _] ),
  read_param2(opposing_team, production_decisions,
            [Quantity, _long_wear, _models, _quality_budget,
            _styling_budget, Improvement_budget, Reject ] ),
  read_param2( average, production_decisions,
            [_, Avg_long_wear, Avg_Models, Avg_quality_budget,
            Avg_styling_budget, _, _ ] ),
  strategy_coefficient( quality, Quality_Strategy, Coeff ),
  New_long_wear is integer(Avg_long_wear * Coeff),
  new_models_wrt_strategy( Avg_Models, Quality_Strategy, New_Models ),
  New_quality_budget is integer(Avg_quality_budget * Coeff),
  New_styling_budget is integer(Avg_styling_budget * Coeff),
  read_param2(opposing_team, labor_decisions,
            [ Workers_employed, _annual_wage, _incentive_pay,
            Worker_productivity ] ),
  read_param2(average, labor_decisions,
            [ _, Avg_annual_wage, Avg_incentive_pay, _ ] ),
  New_annual_wage is integer(Avg_annual_wage * Coeff * 10) / 10,
  New_incentive_pay is integer(Avg_incentive_pay * Coeff * 10) / 10 }
==>
{ write_param2(opposing_team, production_decisions,
            [ Quantity, New_long_wear, New_Models, New_quality_budget,
            New_styling_budget, Improvement_budget, Reject ] ),
  write_param2(opposing_team, labor_decisions, [ Workers_employed,
            New_annual_wage, New_incentive_pay, Worker_productivity ] )
} &
opposing_team : quality_strategy_defined ::= false &
opposing_team : quality_strategy_converted ::= true
--- opposing_teams_make_high_level_decisions.
```

- If last year's `Service_score` is less than 1.0 ( average score ), the short-term service strategy is `reactive`. The new strategy is updated in the `short_term_strategy` metafact as well as in the hidden database.

```
opposing_team : defines_strategy ::= true  &
opposing_team : price_qual_serv_adv_scores(
                        _, _, Service_score, _) ::= true &
{ existing_metafact( opposing_team, short_term_strategy
            ( Price, Quality, _Old_Service, Advertisement), true),
  score_threshold(service, Threshold),
  Score is Service_score + Threshold,
  Score < 1.0 }
==>
opposing_team : short_term_strategy
            ( Price, Quality, reactive, Advertisement ) ::= true &
opposing_team : service_strategy_defined ::= true &
{ write_param2(opposing_team, short_term_strategy,
            [Price, Quality, reactive, Advertisement ] ) }
--- opposing_teams_make_high_level_decisions.
```

- If last year's Service_score is greater than or equal to 1.0 ( average score ), the short-term service strategy is conservative or aggressive. The new strategy is updated in the short_term_strategy metafact as well as in the hidden database.

```
opposing_team : defines_strategy ::= true  &
opposing_team : price_qual_serv_adv_scores(
                        _, _, Service_score, _) ::= true &
{ existing_metafact( opposing_team, short_term_strategy
            ( Price, Quality, _Old_Service, Advertisement), true),
  score_threshold(service, Threshold),
  Score is Service_score + Threshold,
  Score >= 1.0,
  random_member(New_Service_Stgy, [conservative, aggressive]) }
==>
opposing_team : short_term_strategy
            (Price, Quality, New_Service_Stgy, Advertisement ) ::=true&
opposing_team : service_strategy_defined ::= true &
{ write_param2(opposing_team, short_term_strategy,
            [Price, Quality, New_Service_Stgy, Advertisement ] ) }
--- opposing_teams_make_high_level_decisions.
```

- Conversion of the service strategy to quantitative decisions for Dealers, Service_rep and Delivery.

```
opposing_team : service_strategy_defined ::= true &
{ read_param2(opposing_team, short_term_strategy,
            [ _, _, Service_Strategy, _] ),
  read_param2(opposing_team, market_decisions,
            [Price, Adv_budget, _Dealers, Rebates,
            _Service_rep, _Delivery, Sales_forecast ] ),
  read_param2( average, market_decisions,
            [_, _, Avg_Dealers, _, Avg_Service_rep, Avg_Delivery, _ ] ),
  strategy_coefficient( service, Service_Strategy, Coeff ),
  Inv_Coeff is 1/Coeff,
  New_Dealers is integer(Avg_Dealers * Coeff),
  Avg_Dealers_per_serv_rep is integer(Avg_Dealers / Avg_Service_rep),
  New_Dealers_per_serv_rep is Avg_Dealers_per_serv_rep * Inv_Coeff,
  New_Service_rep is integer(New_Dealers / New_Dealers_per_serv_rep ),
  new_delivery_wrt_strategy(Avg_Delivery,Service_Strategy,New_Delivery )
}
==>
{ write_param2(opposing_team, market_decisions,
            [ Price, Adv_budget, New_Dealers, Rebates,
            New_Service_rep, New_Delivery, Sales_forecast ] ) } &
opposing_team : service_strategy_defined ::= false &
opposing_team : service_strategy_converted ::= true
--- opposing_teams_make_high_level_decisions.
```

- If last year's Advertisement_score is less than 1.0 ( average score ), the short-term advertisement strategy is reactive. The new strategy is updated in the short_term_strategy metafact as well as in the hidden database.

```
opposing_team : defines_strategy ::= true  &
opposing_team : price_qual_serv_adv_scores(
                        _, _, _, Advertisement_score) ::= true &
{ existing_metafact( opposing_team, short_term_strategy
            ( Price, Quality, Service, _Old_Advertisement), true),
  score_threshold(advertisement, Threshold),
  Score is Advertisement_score + Threshold,
  Score < 1.0 }
```

```
==>
opposing_team : short_term_strategy
            ( Price, Quality, Service, reactive) ::= true &
opposing_team : advertisement_strategy_defined ::= true &
{ write_param2(opposing_team, short_term_strategy,
            [Price, Quality, Service, reactive ] ) }
--- opposing_teams_make_high_level_decisions.
```

- If last year's Advertisement_score is greater than or equal to 1.0 ( average score ), the short-term advertisement strategy is conservative or aggressive. The new strategy is updated in the short_term_strategy metafact as well as in the hidden database.

```
opposing_team : defines_strategy ::= true &
opposing_team : price_qual_serv_adv_scores(
                        _, _, _, Advertisement_score) ::= true &
{ existing_metafact( opposing_team, short_term_strategy
            ( Price, Quality, Service, _Old_Advertisement), true),
  score_threshold(advertisement, Threshold),
  Score is Advertisement_score + Threshold,
  Score >= 1.0,
  random_member(New_Advertisement_Stgy, [conservative, aggressive]) }
==>
opposing_team : short_term_strategy
            ( Price, Quality, Service, New_Advertisement_Stgy ):=true&
opposing_team : advertisement_strategy_defined ::= true &
{ write_param2(opposing_team, short_term_strategy,
            [Price, Quality, Service, New_Advertisement_Stgy ] ) }
--- opposing_teams_make_high_level_decisions.
```

- Conversion of the advertisement strategy to quantitative decisions for Adv_budget.

```
opposing_team : advertisement_strategy_defined ::= true &
{ read_param2(opposing_team, short_term_strategy,
            [ _, _, _, Adv_Strategy ] ),
  read_param2( opposing_team, market_decisions,
            [Price, _Adv_budget, Dealers, Rebates,
            Service_rep, Delivery_time, Sales_forecast] ),
  read_param2( average, market_decisions,
            [_, Avg_Adv_budget, _, _, _, _, _] ),
  strategy_coefficient( advertisement, Adv_Strategy, Coeff ),
  New_Adv_budget is integer(Avg_Adv_budget * Coeff ) }
==>
{ write_param2(opposing_team, market_decisions,
            [ Price, New_Adv_budget, Dealers, Rebates,
            Service_rep, Delivery_time, Sales_forecast ] ) } &
opposing_team : advertisement_strategy_defined ::= false &
opposing_team : advertisement_strategy_converted ::= true
--- opposing_teams_make_high_level_decisions.
```

- After all the qualitative decisions have been converted to quantitative decisions, estimations are updated.

```
opposing_team : quality_strategy_converted ::= true &
opposing_team : service_strategy_converted ::= true &
opposing_team : advertisement_strategy_converted ::= true &
opposing_team : price_strategy_converted ::= true
==>
{ update_estimations(opposing_team, decision_making) } &
opposing_team : estimations_updated ::= true &
opposing_team : quality_strategy_converted ::= false &
```

```
opposing_team : service_strategy_converted ::= false &
opposing_team : advertisement_strategy_converted ::= false &
opposing_team : price_strategy_converted ::= false
--- opposing_teams_make_high_level_decisions.
```

- After the estimations have been updated, the production `Quantity` decision is made considering the `Reject` rate and `Pairs_sold` estimations and the `Capacity` of the plant.

```
opposing_team : estimations_updated ::= true &
opposing_team : inventory_from_last_year(Inventory_from_last_year,
                  _Mfg_cost, _Avg_models, _Avg_quality) ::= true &
{ read_param2( opposing_team, production_decisions,
            [_Quantity, Long_wear, Models, Quality_budget,
            Styling_budget, Improvement_budget, Reject ] ),
  read_param2( opposing_team, market_decisions,
            [_Price, _Adv_budget, _Dealers, _Rebates,
            _Service_rep, _Delivery_time, Estimated_Pairs_sold] ),
  Quantity_without_inventory is
            integer(Estimated_Pairs_sold/(1-(Reject/100))),
  New_Quantity is Quantity_without_inventory - Inventory_from_last_year,
  production_capacity(opposing_team,Capacity),
  smallest( [ Capacity, New_Quantity ], Smallest ) }
==>
{ write_param2( opposing_team, production_decisions,
            [Smallest, Long_wear, Models, Quality_budget,
            Styling_budget, Improvement_budget, Reject ] ) } &
opposing_team : number_of_pairs_decided ::= true &
opposing_team : estimations_updated ::= false
--- opposing_teams_make_high_level_decisions.
```

- After the production `Quantity` is determined, the Number of `Workers_employed` decision is made considering the `Worker_productivity` estimation and the production `Quantity` decision.

```
opposing_team : number_of_pairs_decided ::= true &
{ cal_workers_needed(opposing_team, Workers_needed),
  read_param2(opposing_team, labor_decisions,
            [ _Workers_employed, Annual_wage,
            Incentive_pay, Worker_productivity ] ) }
==>
{ write_param2(opposing_team, labor_decisions,
            [ Workers_needed, Annual_wage,
            Incentive_pay, Worker_productivity ] ) } &
opposing_team : number_of_workers_decided ::= true &
opposing_team : number_of_pairs_decided ::= false
--- opposing_teams_make_high_level_decisions.
```

- After all the other decisions have been made, the `Price` is made considering the operational cost of product (`Oper_cost`) and the Price `Strategy`.

```
opposing_team : number_of_workers_decided ::= true &
{ read_param2(opposing_team, short_term_strategy, [Strategy, _, _, _] ),
  read_param2(opposing_team, market_decisions,
            [_Price, Adv_budget, Dealers, Rebates,
            Service_rep, Delivery_time, Sales_forecast] ),
  strategy_coefficient( price, Strategy, Coeff ),
  cal_oper_cost_per_produced(opposing_team, Oper_cost),
  New_Price is integer(Oper_cost * Coeff * 100)/100 }
==>
{ write_param2( opposing_team, market_decisions,
```

```
            [ New_Price, Adv_budget, Dealers, Rebates,
            Service_rep, Delivery_time, Sales_forecast ] ) ) &
opposing_team : price_decided ::= true &
opposing_team : number_of_workers_decided ::= false
--- opposing_teams_make_high_level_decisions.
```

- Determination of `automation_option` decision. It can only be made once in the life-time of the game, therefore this metarule can be triggered if it has not yet been made in the previous years.

```
opposing_team : defines_strategy ::= true &
opposing_team : automation_option(_Year, none) ::= true &
{ random_member(New_Option, [ none, a, b, c, d ]),
  read_param2(opposing_team, automation_expansion_decisions,
            [ _Old_Option, Expansion ] ) }
==>
{ write_param2(opposing_team, automation_expansion_decisions,
            [ New_Option, Expansion ] ) }
--- opposing_teams_make_high_level_decisions.
```

- Determination of `New_capacity_increase` decision. If `Capacity_of_plant` is less than the next year's `Demand_estimation`, then this metarule is triggered.

```
opposing_team : defines_strategy ::= true &
environment : demand_per_company(Demand_per_company) ::= true &
{ production_capacity( opposing_team, Capacity_of_plant),
  random(100,110, Demand_inc_estimation),
  Demand_estimation is Demand_per_company * Demand_inc_estimation / 100,
  Capacity_of_plant < Demand_estimation,
  random_member(New_capacity_increase, [0,1000,2000,3000]),
  New_capacity_increase \= 0,
  read_param2(opposing_team, automation_expansion_decisions,
            [ Option, _Old_capacity_increase ] ) }
==>
{ write_param2(opposing_team, automation_expansion_decisions,
            [ Option, New_capacity_increase ] ) }
--- opposing_teams_make_high_level_decisions.
```

- Determination of the `long_term_financing_amount` considering the previous year's `automation_option` decision.

```
opposing_team : defines_strategy ::= true &
environment : current_year(Current_year) ::= true &
opposing_team : automation_option(Year,Option) ::= true &
{ Year_difference is Current_year - Year,
  Year_difference = 1}
==>
{ auto_cost_setup_prod(Option,Investment_amount,_,_),
  existing_metafact(opposing_team,
      long_term_financing_amount(Old_amount), true),
  New_amount is Investment_amount + Old_amount } &
opposing_team : long_term_financing_amount(New_amount) ::= true
--- opposing_teams_make_high_level_decisions.
```

- Determination of the `long_term_financing_amount` considering the previous year's `expansion` decision.

```
opposing_team : defines_strategy ::= true &
{ last_year_expansion_decision( opposing_team, Expansion_amount ) }
==>
{ existing_metafact(opposing_team,
```

```
      long_term_financing_amount(Old_amount), true),
  New_amount is Expansion_amount + Old_amount } &
opposing_team : long_term_financing_amount(New_amount) ::= true
--- opposing_teams_make_high_level_decisions.
```

- Determination of the Financing_method considering the Amount_to_be_financed by long-term financing methods. This metarule triggers if the Amount is not 0 and after price_decided because the Amount_to_be_financed depends on the Net_income. If Net_income is greater than the amount that needs to be financed by Net_income, New_Dividends are paid to the share holders.

```
opposing_team : price_decided ::= true &
{ existing_metafact(opposing_team, long_term_financing_amount
            (Amount), true),
  Amount > 0,
  debt_to_asset_ratio(opposing_team, Ratio),
  net_income(opposing_team, Net_income),
  Amount_to_be_financed is Amount * Ratio,
  (Amount-Amount_to_be_financed) < Net_income,
  random_member(Financing_method,[new_bond_issuing,new_share_issuing]),
  long_term_finance_decision(opposing_team,
            Financing_method, Amount_to_be_financed),
  shares_out(opposing_team, Shares_out, _ ),
  New_Dividends is
            integer((Net_income-(Amount*(1-Ratio)))/Shares_out*10)/10,
  read_param2(opposing_team, financial_decisions,
            [Short_term, Shares, Bonds, _Old_Dividends] ) }
==>
{ write_param2(opposing_team, financial_decisions,
            [Short_term, Shares, Bonds, New_Dividends] ) } &
opposing_team : long_term_financial_decisions_made ::= true &
opposing_team : long_term_financing_amount(Amount) ::= false
--- opposing_teams_make_high_level_decisions.
```

- This is the case when Net_income is less than the amount that needs to be financed by Net_income. No dividends are paid to the share holders.

```
opposing_team : price_decided ::= true &
{ existing_metafact(opposing_team, long_term_financing_amount
            (Amount), true),
  Amount > 0,
  debt_to_asset_ratio(opposing_team, Ratio),
  net_income(opposing_team, Net_income),
  Amount_to_be_financed is Amount * Ratio,
  (Amount-Amount_to_be_financed) >= Net_income,
  Real_Amount_to_be_financed is Amount - Net_income,
  random_member(Financing_method,[new_bond_issuing,new_share_issuing]),
  long_term_finance_decision(opposing_team, Financing_method,
            Real_Amount_to_be_financed),
  read_param2(opposing_team, financial_decisions,
            [Short_term, Shares, Bonds, _Old_Dividends] ) }
==>
{ write_param2(opposing_team, financial_decisions,
            [Short_term, Shares, Bonds, 0] ) } &
opposing_team : long_term_financial_decisions_made ::= true &
opposing_team : long_term_financing_amount(Amount) ::= false
--- opposing_teams_make_high_level_decisions.
```

- Determination of the New_Dividends when the Amount to be financed by a long-term financing method is 0.

```
opposing_team : price_decided ::= true &
{ existing_metafact(opposing_team, long_term_financing_amount
            (Amount), true),
  Amount = 0,
  shares_out(opposing_team, Shares_out, _ ),
  net_income(opposing_team, Net_income),
  New_Dividends is integer( Net_income / Shares_out * 10) / 10,
  read_param2(opposing_team, financial_decisions,
            [Short_term, Shares, Bonds, _Old_Dividends] ) }
==>
{ write_param2(opposing_team, financial_decisions,
            [Short_term, Shares, Bonds, New_Dividends] ) } &
opposing_team : long_term_financial_decisions_made ::= true &
opposing_team : long_term_financing_amount(Amount) ::= false
--- opposing_teams_make_high_level_decisions.
```

- When the `Cash` amount of the company after all the other decisions have been made is less than 0, there is a need to borrow `New_short_term`.

```
opposing_team : price_decided ::= true &
opposing_team : long_term_financial_decisions_made ::= true &
{ revenues( opposing_team, Revenues ),
  cash( opposing_team, Revenues, Cash),
  Cash < 0,
  New_short_term is -Cash }
==>
{ read_param2(opposing_team, financial_decisions,
            [_Old_short_term, Shares, Bonds, Dividends] ),
  write_param2(opposing_team, financial_decisions,
                        [New_short_term, Shares, Bonds, Dividends] ) } &
opposing_team : price_decided ::= false &
opposing_team : long_term_financial_decisions_made ::= false &
opposing_team : defines_strategy ::= false
--- opposing_teams_make_high_level_decisions.
```

- When the `Cash` amount of the company after all the other decisions have been made is greater than 0, there is no need to get a loan.

```
opposing_team : price_decided ::= true &
opposing_team : long_term_financial_decisions_made ::= true &
{ revenues( opposing_team, Revenues ),
  cash( opposing_team, Revenues, Cash),
  Cash >= 0 }
==>
opposing_team : price_decided ::= false &
opposing_team : long_term_financial_decisions_made ::= false &
opposing_team : defines_strategy ::= false
--- opposing_teams_make_high_level_decisions.
```

- After both `opposing_team_1` and `opposing_team_2` have made all their decisions, simulation switches to the `evaluation` packet. The `state_of_student` component of the PDG is modified.

```
opposing_team_1 : defines_strategy ::= false &
opposing_team_2 : defines_strategy ::= false &
student : waits_for_opposing_teams_decisions(Year) ::= true
==>
modify ( state_of_student <- waits_for_evaluation(Year) )
switch_to 'evaluation'
--- opposing_teams_make_high_level_decisions.
```

### THE 'EVALUATION' PACKET

- The evaluation packet starts by updating the average decisions.

```
student : waits_for_evaluation(_Year) ::= true
==>
{ update_average_decisions } &
student : waiting_score_updating ::= true &
opposing_team_1 : waiting_score_updating ::= true &
opposing_team_2 : waiting_score_updating ::= true
--- evaluation.
```

- The student's scores are updated and the `scores` component of the PDG is virtually modified by using `procedure` construct. After updating the scores, the estimations are replaced with their actual values.

```
student : waiting_score_updating ::= true &
{ update_scores( student, Price_score, Quality_score,
                         Service_score, Advertisement_score ) }
==>
student : procedure (
      scores <- price_qual_serv_adv_scores
                  ( Price_score, Quality_score,
                  Service_score, Advertisement_score ) ) ::= true &
student : price_qual_serv_adv_scores(Price_score, Quality_score,
                         Service_score, Advertisement_score) ::= true &
{ update_estimations( student, evaluation) } &
student : waiting_score_updating ::= false &
student : short_term_debt_modification ::= true
--- evaluation.
```

- The opposing teams' scores are updated and their estimations are replaced with their actual values.

```
opposing_team : waiting_score_updating ::= true &
{ update_scores( opposing_team, Price_score, Quality_score,
                         Service_score, Advertisement_score ) }
==>
opposing_team : price_qual_serv_adv_scores( Price_score, Quality_score,
                         Service_score, Advertisement_score ) ::= true &
{ update_estimations( opposing_team, evaluation) } &
opposing_team : waiting_score_updating ::= false &
opposing_team : short_term_debt_modification ::= true
--- evaluation.
```

- As a result of updating the estimations, the student's company is automatically given a `New_Short_term_loan` to compensate the negative value of `cash`.

```
student : short_term_debt_modification ::= true &
{ revenues( student, Revenues ),
  cash( student, Revenues, Cash),
  Cash < 0 }
==>
{ read_param2(student, financial_decisions,
            [ Old_Short_term_loan, Shares_issued,
            Bond_issue, Dividend_declaration ] ),
  New_Short_term_loan is Old_Short_term_loan - Cash,
  write_param2(student, financial_decisions,
            [ New_Short_term_loan, Shares_issued,
            Bond_issue, Dividend_declaration ] ) } &
student : short_term_debt_modification ::= false &
```

```
student : waiting_company_reseting ::= true
--- evaluation.
```

- The value of cash is greater than 0, therefore the student's company is not automatically given a New_Short_term_loan.

```
student : short_term_debt_modification ::= true &
{ revenues( student, Revenues ),
  cash( student, Revenues, Cash),
  Cash >= 0 }
==>
student : short_term_debt_modification ::= false &
student : waiting_company_reseting ::= true
--- evaluation.
```

- As a result of updating the estimations, the opposing teams' companies having a negative value of cash, are automatically given a New_Short_term_loan.

```
opposing_team : short_term_debt_modification ::= true &
{ revenues( opposing_team, Revenues ),
  cash( opposing_team, Revenues, Cash),
  Cash < 0 }
==>
{ read_param2(opposing_team, financial_decisions,
            [ Old_Short_term_loan, Shares_issued,
            Bond_issue, Dividend_declaration ] ),
  New_Short_term_loan is Old_Short_term_loan - Cash,
  write_param2(opposing_team, financial_decisions,
            [ New_Short_term_loan, Shares_issued,
            Bond_issue, Dividend_declaration ] ) } &
opposing_team : short_term_debt_modification ::= false &
opposing_team : waiting_company_reseting ::= true
--- evaluation.
```

- The opposing teams' companies having positive value of cash, are not automatically given a New_Short_term_loan.

```
opposing_team : short_term_debt_modification ::= true &
{ revenues( opposing_team, Revenues ),
  cash( opposing_team, Revenues, Cash),
  Cash >= 0 }
==>
opposing_team : short_term_debt_modification ::= false &
opposing_team : waiting_company_reseting ::= true
--- evaluation.
```

- The student's company related historical information is updated and the history component of the PDG is virtually modified using the procedure construct.

```
student : waiting_company_reseting ::= true
==>
{ reset_inventories(student,
            Quantity_unsold, Mfg_cost, Avg_models, Avg_quality ),
  all_indicators( student, Revenues, Net_income, Eps, Roe, Cash ),
  reset_bond_rating( student, Bond_rating ),
  times_interest_earned( student, Times_interest),
  reset_bonds(student, All_bonds),
  read_param2(student, financial_decisions, [Short_term_loan, _, _, _]),
  reset_shares_and_capital(student, All_shares_capital),
  acc_retained_earning( student, Net_income, Acc_retained_earn) } &
student : procedure (
```

```
      history <-
            last_year_indicators(Revenues, Net_income, Eps, Roe, Cash) &
            last_year_bond_rating( Bond_rating ) &
            last_year_times_interest_earned( Times_interest ) &
            bonds_issued_so_far( All_bonds ) &
            short_term_loan_from_last_year( Short_term_loan ) &
            shares_and_capital( All_shares_capital ) &
            accumulated_retained_earnings(Acc_retained_earn) &
            inventory_from_last_year(Quantity_unsold,Mfg_cost,
                              Avg_models,Avg_quality) ) ::= true &
student : waiting_company_reseting ::= false &
student : waiting_plant_reseting ::= true
--- evaluation.
```

- The historical information of the opposing teams' companies is updated.

```
opposing_team : waiting_company_reseting ::= true
==>
{ reset_inventories(opposing_team,
            Quantity_unsold, Mfg_cost, Avg_models, Avg_quality ),
  all_indicators( opposing_team, Revenues, Net_income, Eps, Roe, Cash ),
  reset_bond_rating( opposing_team, Bond_rating ),
  times_interest_earned( opposing_team, Times_interest),
  reset_bonds(opposing_team, All_bonds),
  read_param2(opposing_team, financial_decisions,
            [ Short_term_loan, _, _, _ ] ),
  reset_shares_and_capital(opposing_team, All_shares_capital),
  acc_retained_earning( opposing_team, Net_income, Acc_retained_earn)}&
opposing_team : last_year_indicators(
            Revenues, Net_income, Eps, Roe, Cash) ::= true &
opposing_team : last_year_bond_rating( Bond_rating ) ::= true &
opposing_team : last_year_times_interest_earned( Times_interest)::=true&
opposing_team : bonds_issued_so_far( All_bonds ) ::= true &
opposing_team : short_term_loan_from_last_year( Short_term_loan)::=true&
opposing_team : shares_and_capital( All_shares_capital ) ::= true &
opposing_team : accumulated_retained_earnings(Acc_retained_earn)::=true&
opposing_team : inventory_from_last_year(Quantity_unsold,Mfg_cost,
            Avg_models,Avg_quality) ::= true &
opposing_team : waiting_company_reseting ::= false &
opposing_team : waiting_plant_reseting ::= true
--- evaluation.
```

- The student's plant expansion information is updated and the `expansion_status` component of the PDG is virtually modified using the `procedure` construct.

```
student : waiting_plant_reseting ::= true &
{ read_param2(student, automation_expansion_decisions, [ _, Expansion]),
  Expansion \= 0,
  update_expansions(student, Expansion, All_expansions) }
==>
student : procedure (
      expansion_status
            <- expansions_year_cost_capacity(All_expansions) ) ::= true&
student : waiting_plant_reseting ::= false
--- evaluation.
```

- The plant expansion information of the opposing teams is updated.

```
opposing_team : waiting_plant_reseting ::= true &
{ read_param2(opposing_team, automation_expansion_decisions,
            [ Automation_option, Expansion ] ),
  Expansion \= 0,
```

```
  update_expansions(opposing_team, Expansion, All_expansions) }
==>
{ write_param2(opposing_team, automation_expansion_decisions,
            [ Automation_option, 0 ] ) } &
opposing_team : expansions_year_cost_capacity(All_expansions) ::= true &
opposing_team : waiting_plant_reseting ::= false
--- evaluation.
```

- The student's plant automation information is updated and the `automation_status` component of the PDG is virtually modified using the `procedure` construct.

```
student : waiting_plant_reseting ::= true &
environment : current_year(Year) ::= true &
{ read_param2(student, automation_expansion_decisions,
            [ Automation_option, Expansion ] ),
  Automation_option \= none,
  write_param2(student, automation_expansion_decisions,
            [ none, Expansion ] ) }
==>
student : procedure (
      automation_status <-
            automation_option(Year, Automation_option) ) ::= true &
student : waiting_plant_reseting ::= false
--- evaluation.
```

- The plant automation information of the opposing teams is updated.

```
opposing_team : waiting_plant_reseting ::= true &
environment : current_year(Year) ::= true &
{ read_param2(opposing_team, automation_expansion_decisions,
            [ Automation_option, Expansion ] ),
  Automation_option \= none }
==>
{ write_param2(opposing_team, automation_expansion_decisions,
            [ none, Expansion ] ) } &
opposing_team : automation_option(Year, Automation_option) ::= true &
opposing_team : waiting_plant_reseting ::= false
--- evaluation.
```

- The student's other plant related information is updated and the `other_status` component of the PDG is virtually modified using the `procedure` construct.

```
student : waiting_plant_reseting ::= true &
student : cumulative_quality_expenditure(Old_cumul_qual_budget)::= true&
student : cumulative_improvement_expenditure(
                        Old_cumul_improv_budget) ::= true &
student : cumulative_advertisement_expenditure(
                        Old_cumul_adv_budget) ::= true &
{ read_param2(student, production_decisions,
            [ _, _, _, Quality_budget, _, Improvement_budget, _ ] ),
  Cumulative_quality_budget is Old_cumul_qual_budget + Quality_budget,
  Cumulative_improv_budget is
            Old_cumul_improv_budget + Improvement_budget,
  read_param2(student, market_decisions,
            [ _, Advertising_budget, _, _, _, _, _ ] ),
  Cumulative_adv_budget is Old_cumul_adv_budget + Advertising_budget,
  read_param2(student, labor_decisions, [ _, Annual_wage, _, _ ] ) }
==>
student : procedure (
      other_status <-
            annual_wage_for_last_year(Annual_wage) &
            cumulative_quality_expenditure(Cumulative_quality_budget) &
```

```
            cumulative_advertisement_expenditure(Cumulative_adv_budget)&
            cumulative_improvement_expenditure(Cumulative_improv_budget)
      ) ::= true &
student : waiting_plant_reseting ::= false
--- evaluation.
```

- Other plant related information of the opposing teams is updated.

```
opposing_team : waiting_plant_reseting ::= true &
opposing_team : cumulative_quality_expenditure(
                        Old_cumul_qual_budget) ::= true &
opposing_team : cumulative_improvement_expenditure(
                        Old_cumul_improv_budget) ::= true &
opposing_team : cumulative_advertisement_expenditure(
                        Old_cumul_adv_budget) ::= true &
{ read_param2(opposing_team, production_decisions,
            [ _, _, _, Quality_budget, _, Improvement_budget, _ ] ),
  Cumulative_quality_budget is Old_cumul_qual_budget + Quality_budget,
  Cumulative_improv_budget is
            Old_cumul_improv_budget + Improvement_budget,
  read_param2(opposing_team, market_decisions,
            [ _, Advertising_budget, _, _, _, _, _ ] ),
  Cumulative_adv_budget is Old_cumul_adv_budget + Advertising_budget,
  read_param2(opposing_team, labor_decisions, [ _, Annual_wage, _, _]) }
==>
opposing_team : annual_wage_for_last_year(Annual_wage) ::= true &
opposing_team : cumulative_quality_expenditure
            (Cumulative_quality_budget )::= true &
opposing_team : cumulative_advertisement_expenditure
            (Cumulative_adv_budget) ::= true &
opposing_team : cumulative_improvement_expenditure
            (Cumulative_improv_budget) ::= true &
opposing_team : waiting_plant_reseting ::= false
--- evaluation.
```

- The automatically updated short-term loan decision and the actual outside factors are virtually updated in the `decisions` component of the PDG.

```
student : waiting_plant_reseting ::= true
==>
{ read_param2(student, production_decisions,
            [ Quantity, Long_wear, Models, Quality_budget,
            Styling_budget, Improvement_budget, Reject ] ),
  read_param2(student, labor_decisions,
            [ Workers_employed, Annual_wage,
            Incentive_pay, Worker_productivity ] ),
  read_param2(student, automation_expansion_decisions,
            [ Automation_option, Expansion ] ),
  read_param2(student, market_decisions,
            [ Wholesale_price, Advertising_budget, Dealers, Rebates,
            Service_rep, Delivery_time, Sales_forecast ] ),
  read_param2(student, financial_decisions,
            [ Short_term_loan, Shares_issued,
            Bond_issue, Dividend_declaration ] ) } &
student : procedure (
      decisions <-
            production_decisions( Quantity, Long_wear, Models,
                  Quality_budget, Styling_budget, Improvement_budget,
                  Reject ) &
            labor_decisions( Workers_employed, Annual_wage,
                  Incentive_pay, Worker_productivity ) &
            automation_expansion_decisions( Automation_option,
```

```
            Expansion ) &
      market_decisions( Wholesale_price, Advertising_budget,
            Dealers, Rebates, Service_rep, Delivery_time,
            Sales_forecast ) &
      financial_decisions( Short_term_loan, Shares_issued,
            Bond_issue, Dividend_declaration ) ) ::= true
--- evaluation.
```

- After updating the plant related information, the simulation switches to the `environment_resetting` packet. The `state_of_student` component of the PDG is modified.

```
student : waiting_plant_reseting ::= false &
opposing_team_1 : waiting_plant_reseting ::= false &
opposing_team_2 : waiting_plant_reseting ::= false &
environment : current_year(Year) ::= true &
{ save_all_modifications(student) }
==>
modify ( state_of_student <- waits_for_environment_resetting(Year) )
switch_to 'environment_resetting'
--- evaluation.
```

### THE 'ENVIRONMENT RESETTING' PACKET

- The teams are getting ready for their `stock_price_modification`.

```
student : waits_for_environment_resetting(_Year) ::= true
==>
student : stock_price_modification ::= true &
opposing_team_1 : stock_price_modification ::= true &
opposing_team_2 : stock_price_modification ::= true
--- environment_resetting.
```

- `stock_price_modification` of the student's company.

```
student : stock_price_modification ::= true &
environment : stock_price(student,Price) ::= true &
student : last_year_indicators(_,_,Roe,Eps,_) ::= true &
{ calculate_average_roe( Avg_Roe ),
  calculate_average_eps( Avg_Eps ),
  Stock_price_factor is ( Roe / Avg_Roe + Eps / Avg_Eps ) / 2,
  New_Price is integer(Price * Stock_price_factor * 10) / 10 }
==>
student : stock_price_modification ::= false &
environment : stock_price(student,Price) ::= false &
environment : stock_price(student,New_Price) ::= true
--- environment_resetting.
```

- `stock_price_modification` of the opposing team's companies.

```
opposing_team : stock_price_modification ::= true &
opposing_team : last_year_indicators(_,_,Roe,Eps,_) ::= true &
{ existing_metafact(environment,
            stock_price(opposing_team, Price), true),
  calculate_average_roe( Avg_Roe ),
  calculate_average_eps( Avg_Eps ),
  Stock_price_factor is ( Roe / Avg_Roe + Eps / Avg_Eps ) / 2,
  New_Price is integer(Price * Stock_price_factor * 10) / 10 }
==>
opposing_team : stock_price_modification ::= false &
environment : stock_price(opposing_team, Price) ::= false &
environment : stock_price(opposing_team,New_Price) ::= true
--- environment_resetting.
```

- The `current_year` metafact of the environment is updated for the following year.

```
environment : current_year(Year) ::= true &
{ New_year is Year + 1 }
==>
environment : current_year(New_year) ::= true
--- environment_resetting.
```

- The `long_wear_price` and `normal_wear_price` metafacts of the environment are updated for the following year.

```
environment : long_wear_price(Long_price) ::= true &
environment : normal_wear_price(Normal_price) ::= true &
{ read_param2( average, production_decisions,
            [_,Avg_long_percent,_,_,_,_,_] ),
  long_wear_percentage_threshold(Threshold_long_percent),
  Avg_long_percent > Threshold_long_percent }
==>
{ random(0,10,Long_increase_percent),
  random(0,10,Normal_decrease_percent),
```

```
  Long_factor is 1 + Long_increase_percent / 100,
  Normal_factor is 1 - Normal_decrease_percent / 100,
  New_Long_price is integer(Long_price * Long_factor * 10)/10,
  New_Normal_price is integer(Normal_price * Normal_factor * 10)/10 } &
environment : long_wear_price(New_Long_price) ::= true &
environment : normal_wear_price(New_Normal_price) ::= true
--- environment_resetting.
```

- The demand_per_company metafact of the environment is updated for the following year.

```
environment : demand_per_company(Demand_per_company) ::= true &
{ random(100, 110, Demand_inc_factor),
  New_Demand_per_company is
            integer(Demand_per_company * Demand_inc_factor / 100) }
==>
environment : demand_per_company(New_Demand_per_company) ::= true
--- environment_resetting.
```

- After the environmental metafacts have been updated, the simulation switches to the student_decision_menu packet. The state_of_student component of the PDG is modified.

```
student : waits_for_environment_resetting(_Year) ::= true &
environment : current_year(New_Year) ::= true
==>
modify ( state_of_student <- makes_decision_year_round(New_Year,1) )
switch_to 'student_decision_menu'
--- environment_resetting.
```

## APPENDIX C. PREDICATES USED IN METARULE WINDOWS

These are the predicates used in the metarule windows of the BSG application. The terms that appear in the `typewriter font` represent the predicate name, the terms that appear in *italics* represent the inputs and the terms that appear in ***italics bold*** represent the outputs.

`acc_retained_earning(` *Team*, *Net_income*, ***Accumulated_retained_earnings*** )

Given a team's name and net income, returns the accumulated retained earning amount of that team.

`all_indicators(` *Team*, ***Revenues***, *Net_income*, ***Eps***, ***Roe***, ***Cash*** )

Given a team's name, returns the revenues, net income, earnings per share (eps), return on equity (roe) and cash indicators of that team.

`auto_cost_setup_prod(` *Option*, ***Cost***, ***Setup_cost_gain_ratio***, ***Capacity*** )

Given an automation option, returns the cost of the automation, the set-up cost gain ratio and the capacity increase resulting from that automation option.

`cal_oper_cost_per_produced(` *Team*, ***Oper_cost***)

Given a team's name, returns the operational cost per product of the company of that team.

`cal_workers_needed(` *Team*, ***Workers_needed***)

Given a team's name, returns the number of workers that need to be hired.

`calculate_average_eps(` ***Avg_Eps*** )

Returns the average earnings per share (eps) indicator of all the companies.

`calculate_average_roe(` ***Avg_Roe*** )

Returns the average return on equity (eps) indicator of all the companies.

`cash(` *Team*, *Revenues*, ***Cash***)

Given a team's name and net income, returns the cash amount of that team.

`convert2string_all(` *List_of_atoms*, ***List_of_string*** )

Given a list of atoms, each atom is converted to a string and returned in a list of strings.

`debt_to_asset_ratio(` *Team*, ***Ratio***)

Given a team's name, returns the debt to asset ratio of that team.

`display_all_screens`

Displays the production, the labor, automation-expansion, the marketing and the financial screens in a window.

`display_automation_expansion_screen`

Displays the automation-expansion screen in a window.

`display_financial_screen`

Displays the financial screen in a window.

`display_labor_screen`

Displays the labor screen in a window.

`display_marketing_screen`

Displays the marketing screen in a window.

`display_production_screen`

Displays the production screen in a window.

`existing_metafact(` *Team*, *Fact*, *Truth_value* )

Succeeds if the given fact that belongs to the given team's name exist with the given truth value.

`give_warning(` *Message*, *Size* )

Display the given message in a window with a given size.

`last_year_expansion_decision(` *Team*, ***Expansion_amount*** )

Given a team's name, returns the expansion decision made in the previous year.

`long_term_finance_decision(`*Team*, *Financing_method*, *Amount_to_be_financed*)

Given a team's name and financing method makes the long-term financing decision considering the financing method and the given amount to be financed.

`long_wear_percentage_threshold(`***Threshold_long_percent***)

Returns the long-wear threshold percentage.

`minimum_annual_wage(`***Minimum_annual_wage***)

Returns the minimum annual wage allowed in the game.

`net_income(` *Team*, ***Net_income***)

Given a team's name, returns the net income of that team's company.

`new_delivery_wrt_strategy(` *Avg_Delivery*, *Service_Strategy*, ***New_Delivery***)

Given the average delivery time decision of all the companies and a service strategy decision, returns a new delivery time decision.

`new_models_wrt_strategy(` *Avg_Models*, *Quality_Strategy*, ***New_models*** )

Given the average number of models decisions of all the companies and a quality strategy decision, returns a new number of models decision.

`new_rebates_wrt_strategy(` *Avg_Rebates*, *Price_Strategy*, ***New_Rebates***)

Given the average rebates decisions of all the companies and a price strategy decision, returns a new rebates decision.

`production_capacity(` *Team*, ***Capacity***)

Given a team's name, returns the production capacity of that team's company.

`random(` *Lower*, *Upper*, ***Random_value***)

Given a lower limit and an upper limit, returns a value within that range randomly.

`random_member(` ***Random_element***, *List* )

Given a list of elements, returns an element of that list randomly.

`read_param2(` *Param_1*, *Param_2*, ***Value*** )

Given two parameters, returns the value stored in the hidden database corresponding to those parameters.

`reset_bond_rating(` *Team*, ***Bond_rating*** )

Given a team's name, returns the bond rating corresponding to that team.

`reset_bonds(` *Team*, ***All_bonds***)

Given a team's name, returns a list of bonds issued until the current year including the bonds issued in the current year.

`reset_inventories(` *Team*, ***Products_unsold***,***Mfg_cost***,***Avg_models***,***Avg_quality***)

Given a team's name, returns the number of products unsold, the manufacturing cost per product, the average number of models in those products and the average quality of those products.

`reset_shares_and_capital(` *Team*, ***All_shares_capital***)

Given a team's name, returns a list of shares issued until the current year including the shares issued in the current year.

`revenues(` *Team*, ***Revenues*** )

Given a team's name, returns the revenues indicator of that team's company.

`save_all_modifications(` *Supported_team* )

Given the supported team's name, incorporates all the virtual modifications of the PDG with the modifications appearing in the modification metarule.

`score_threshold(` *Strategy_category*, ***Threshold***)

Given a strategy category (price, quality, service or advertisement), returns the threshold value corresponding to that category that will be used in determining the new strategy.

`shares_out(` *Team*, ***Shares_out***, ***Total_capital*** )

Given a team's name, returns the total number of shares outstanding and the total capital earned by issuing those shares.

`smallest(` *List_of_numbers*, ***Smallest*** )

Given a list of numbers, returns the smallest number in that list.

`strategy_coefficient(` *Strategy_category*, *Strategy*, ***Coeff*** )

Given a strategy category (price, quality, service or advertisement) and a strategy (conservative, aggressive or reactive) corresponding to that category, returns the coefficient corresponding to that category and to that strategy that will be used in the conversion of qualitative decisions into quantitative decisions.

`times_interest_earned(` *Team*, ***Times_interest***)

Given a team's name, returns the times-interest-earned ratio of that team's company.

`update_average_decisions`

Update the averages of the quantitative decisions of all the companies in the hidden database.

`update_estimations(` *Team*, *Phase*)

Given a team's name and a phase (decision_making or evaluation) updates the estimation values of the reject rate, the worker productivity and the sale amount. If the phase is decision-making, the sale amount is determined considering the selected strategies. If the phase is evaluation, it is determined considering the current year average decisions of all the teams.

`update_expansions(` *Team*, *Expansion*, ***All_expansions***)

Given a team's name and the current year's expansion decision, returns an updated list of all expansion decisions made so far.

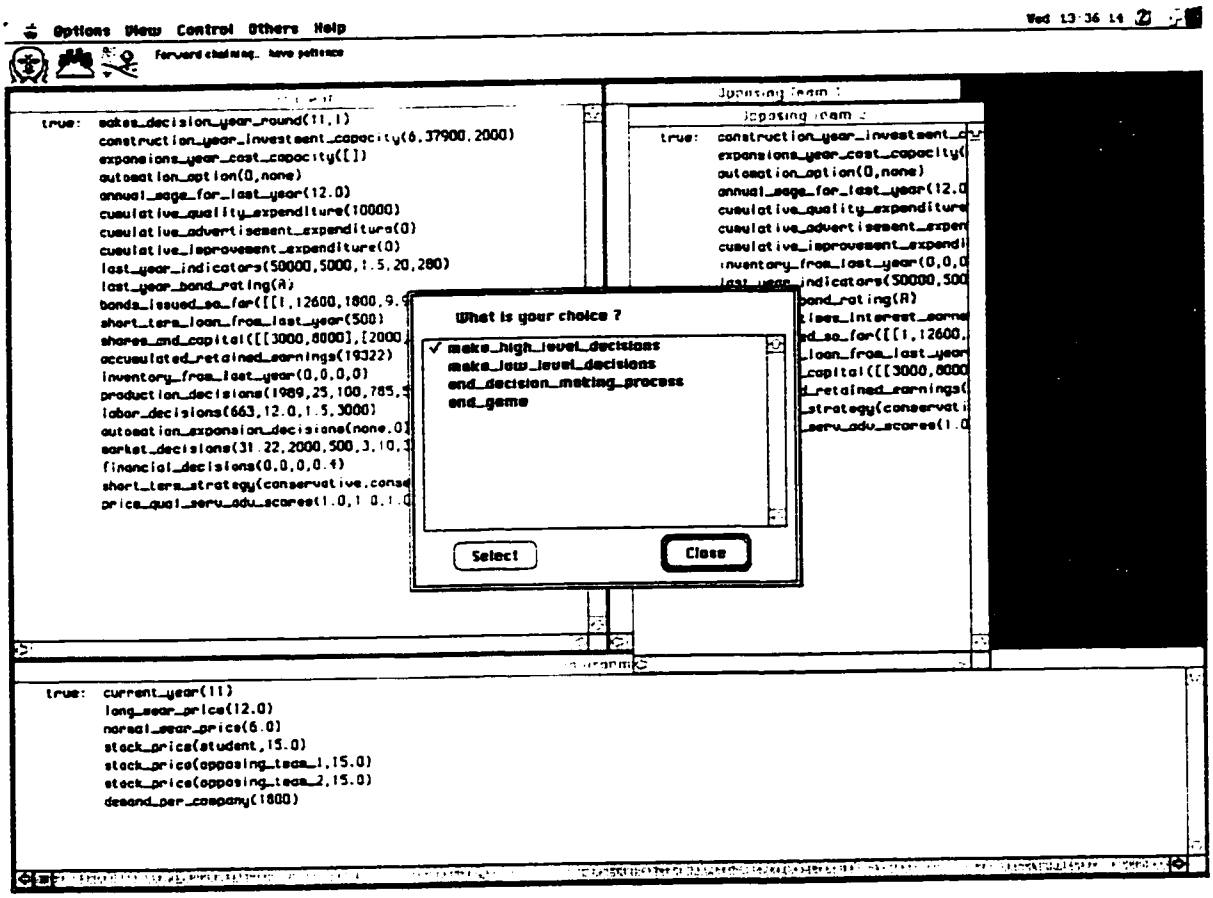
`update_scores(`*Team*, ***Price_score***, ***Quality_score***, ***Service_score***, ***Advertisement_score***`)`

Given a team's name, returns the price, the quality, the service and the advertisement scores.

`write_param2(` *Param_1*, *Param_2*, ***Value***`)`

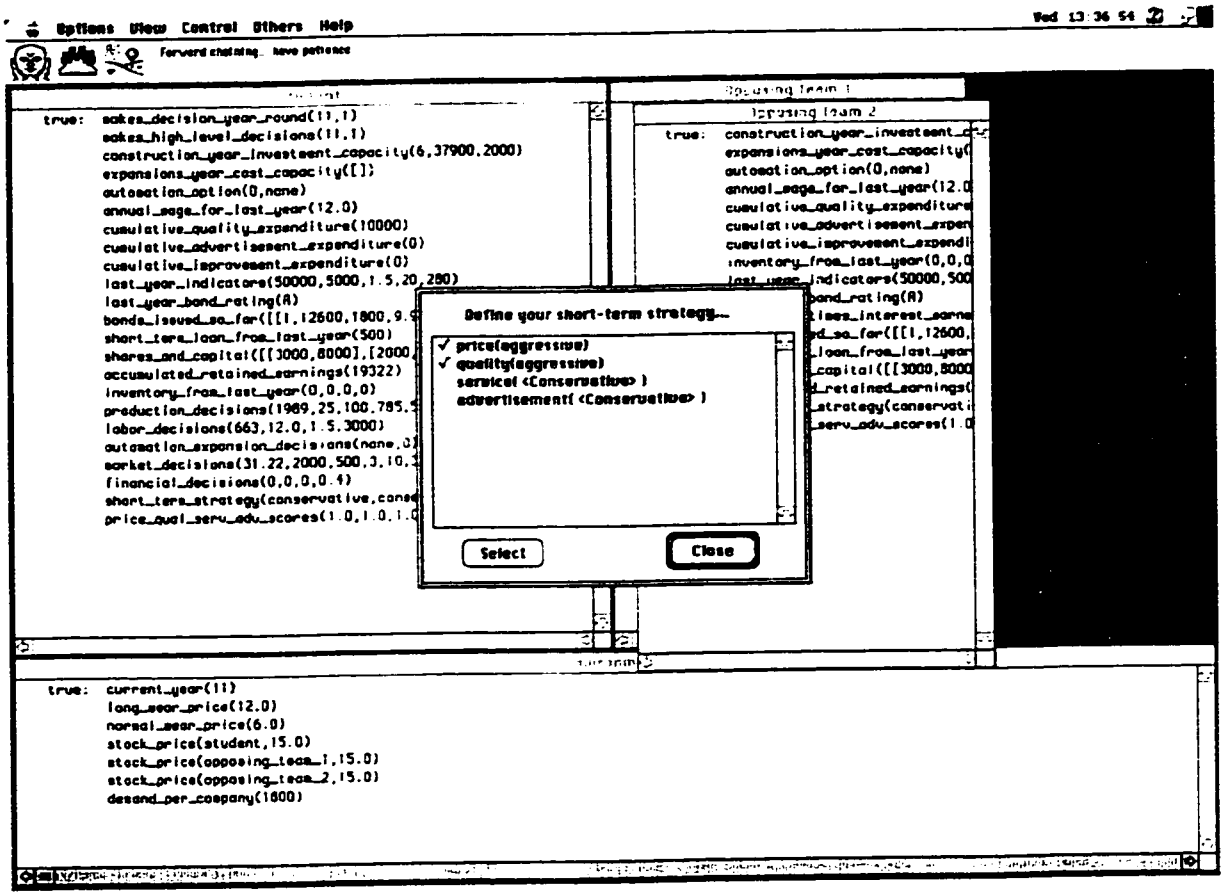
Given two parameters, replace the value stored in the hidden database corresponding to those parameters with the given value.

# APPENDIX D. SNAPSHOTS FROM THE BSG APPLICATION




## Appendix D.1. THE MAIN SELECTION MENU

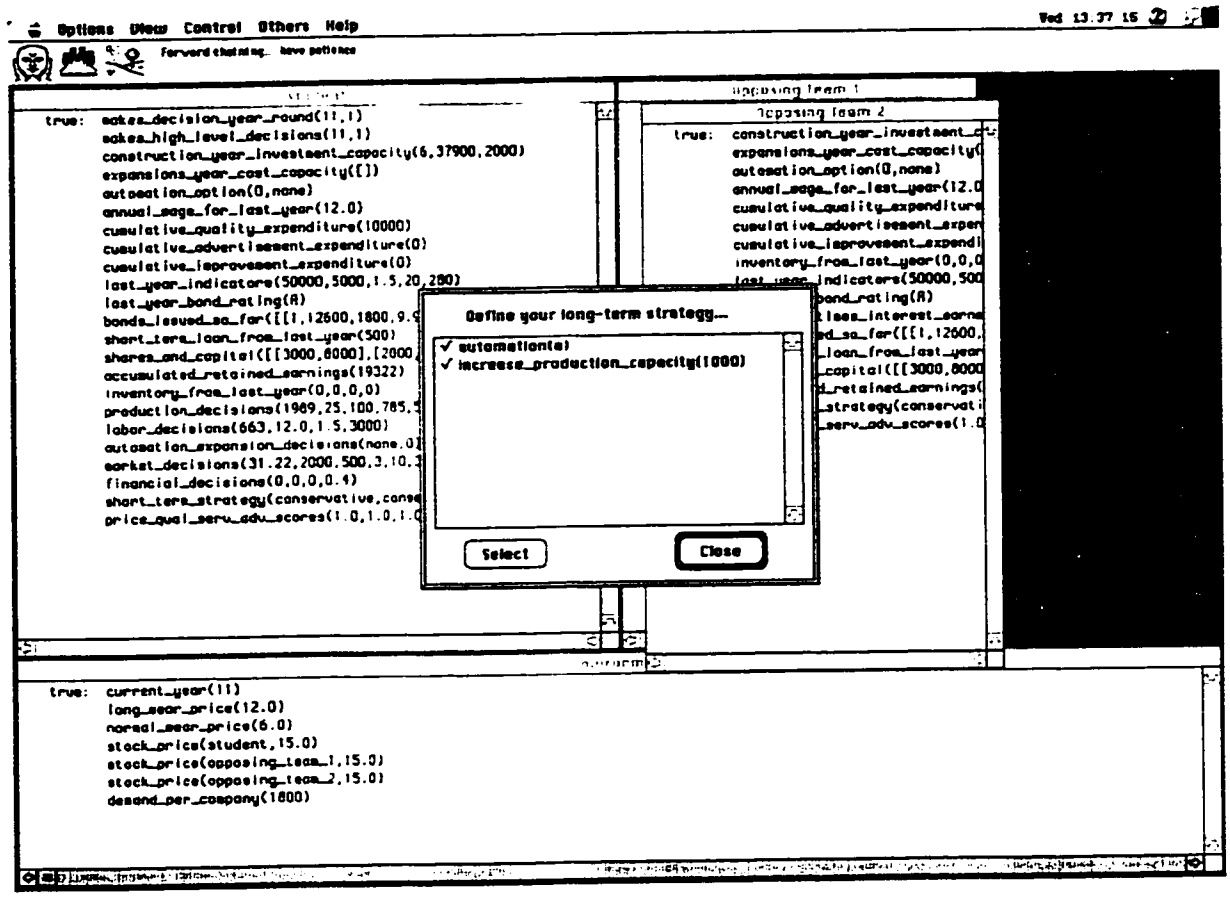
The user wants to continue with the high-level decision-making process.




## Appendix D.2. DEFINITION OF A SHORT-TERM STRATEGY

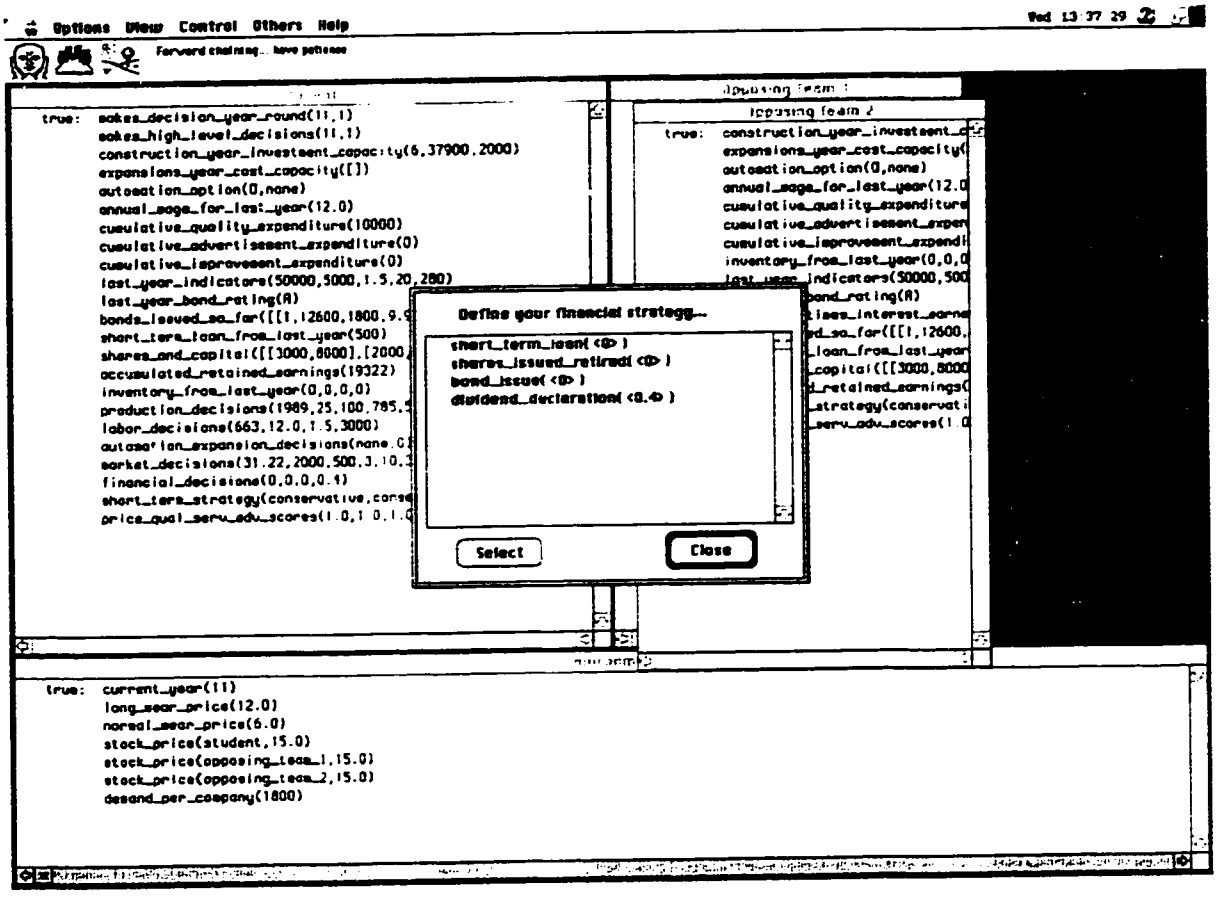
The user wants to be aggressive in price and quality and conservative in service and advertisement.

The previous high-level decisions appear as the default values in the selection menu.




## Appendix D.3. DEFINITION OFA LONG-TERM STRATEGY

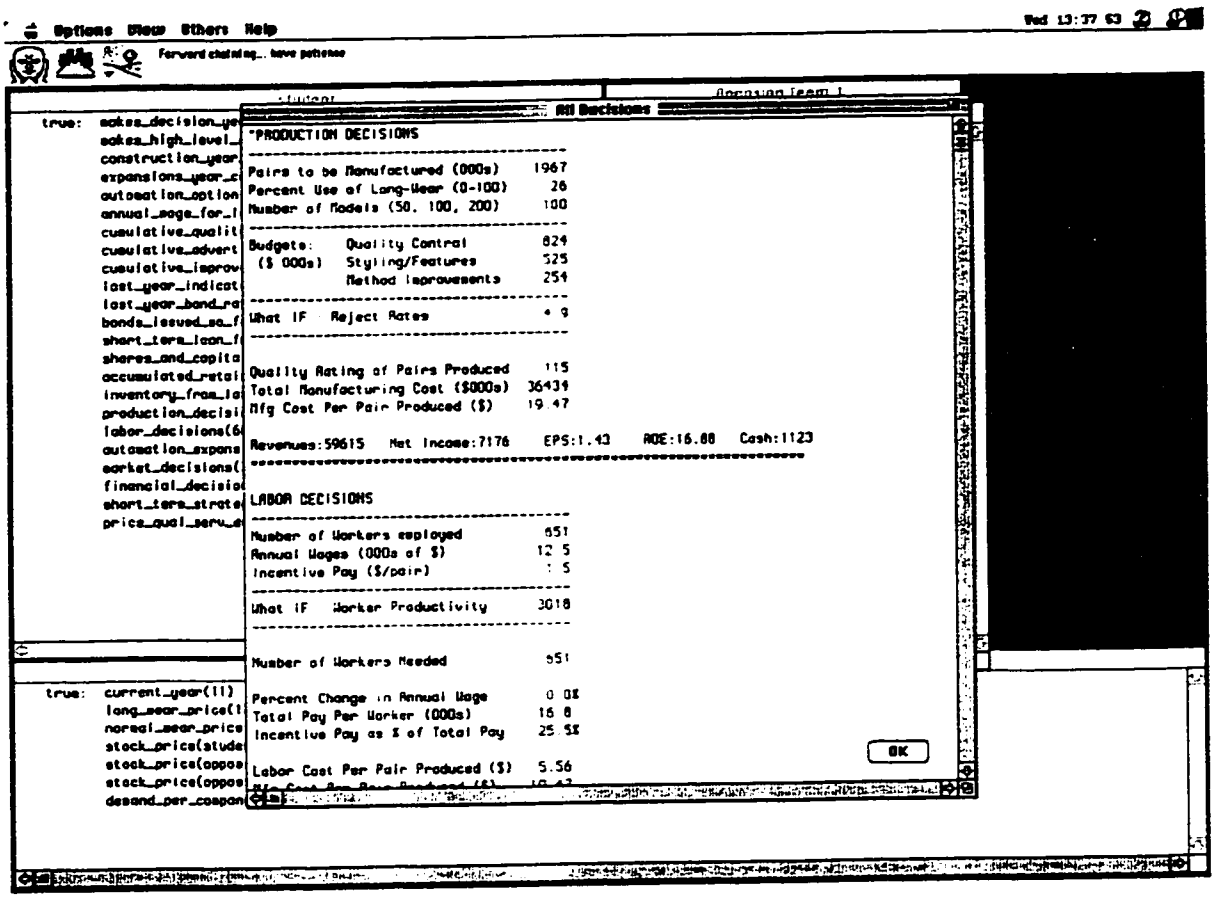
The user selects the automation option "a" and wants to increase the production capacity by 1000.




## Appendix D.4. DEFINITION OF A FINANCIAL STRATEGY

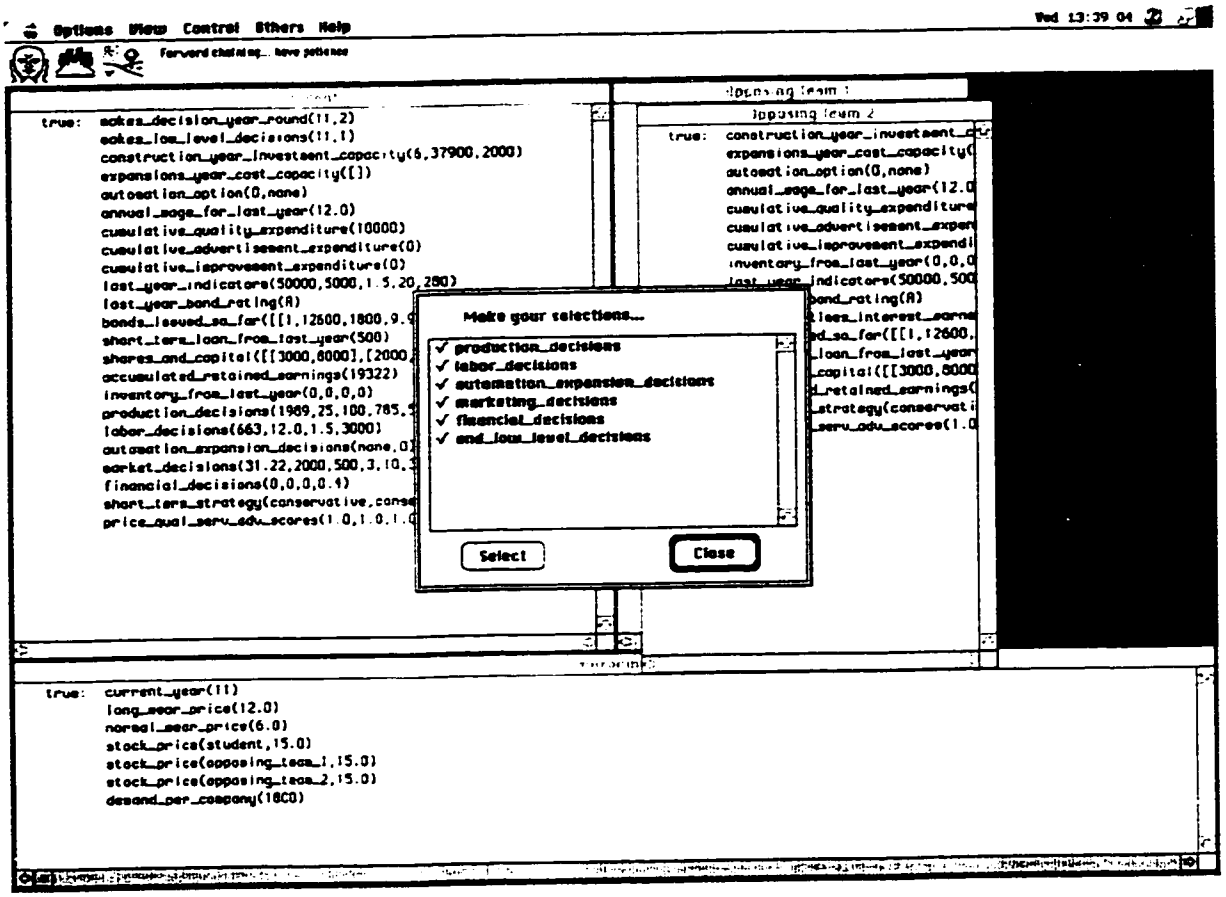
No decision has been made yet.

The previous financial decisions appear as the default values in the selection menu.



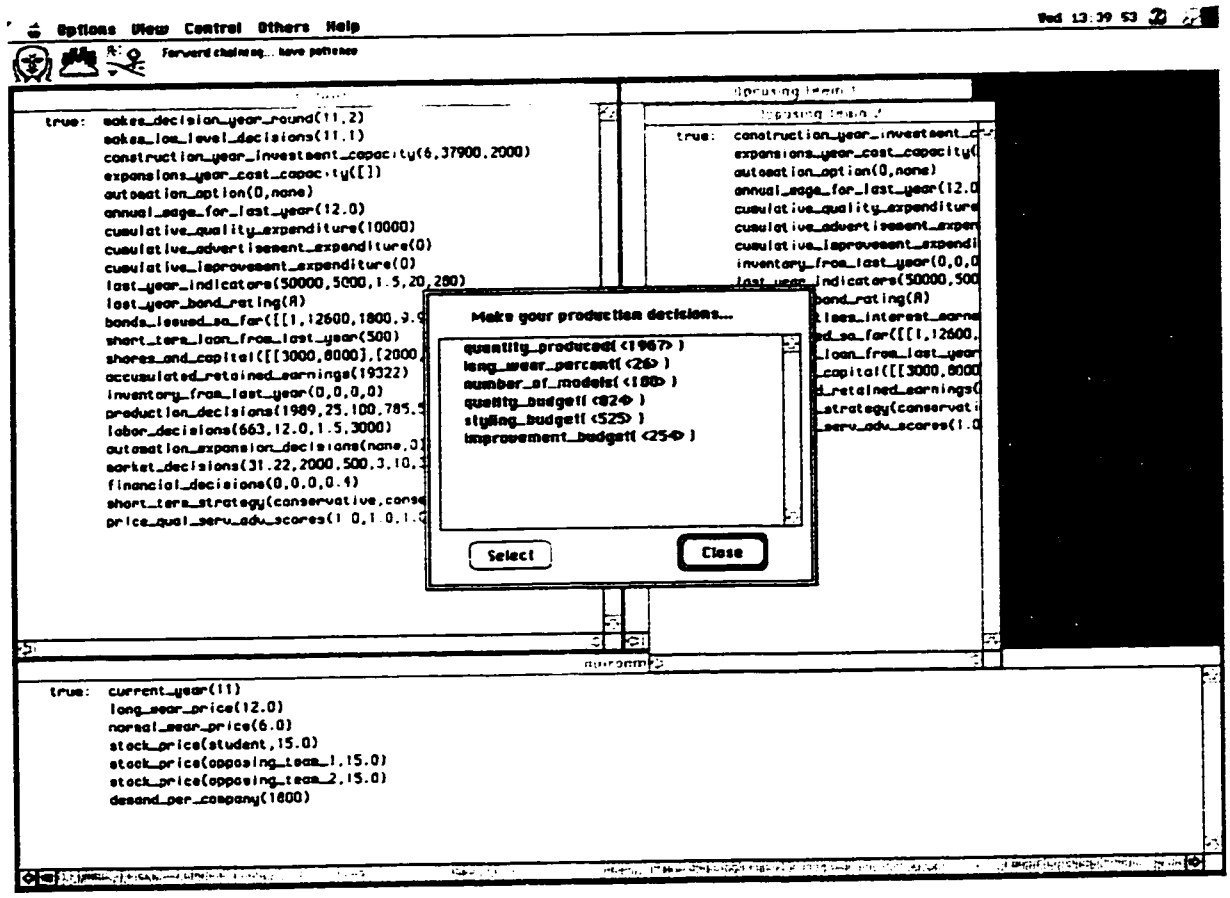

## Appendix D.5. ALL DECISIONS SCREEN

After defining the short-term, the long-term and the financial strategies, the selected strategies are converted into their corresponding quantitative decisions and the estimated financial indicators are displayed to the user, as well as all the quantitative decision conversions.



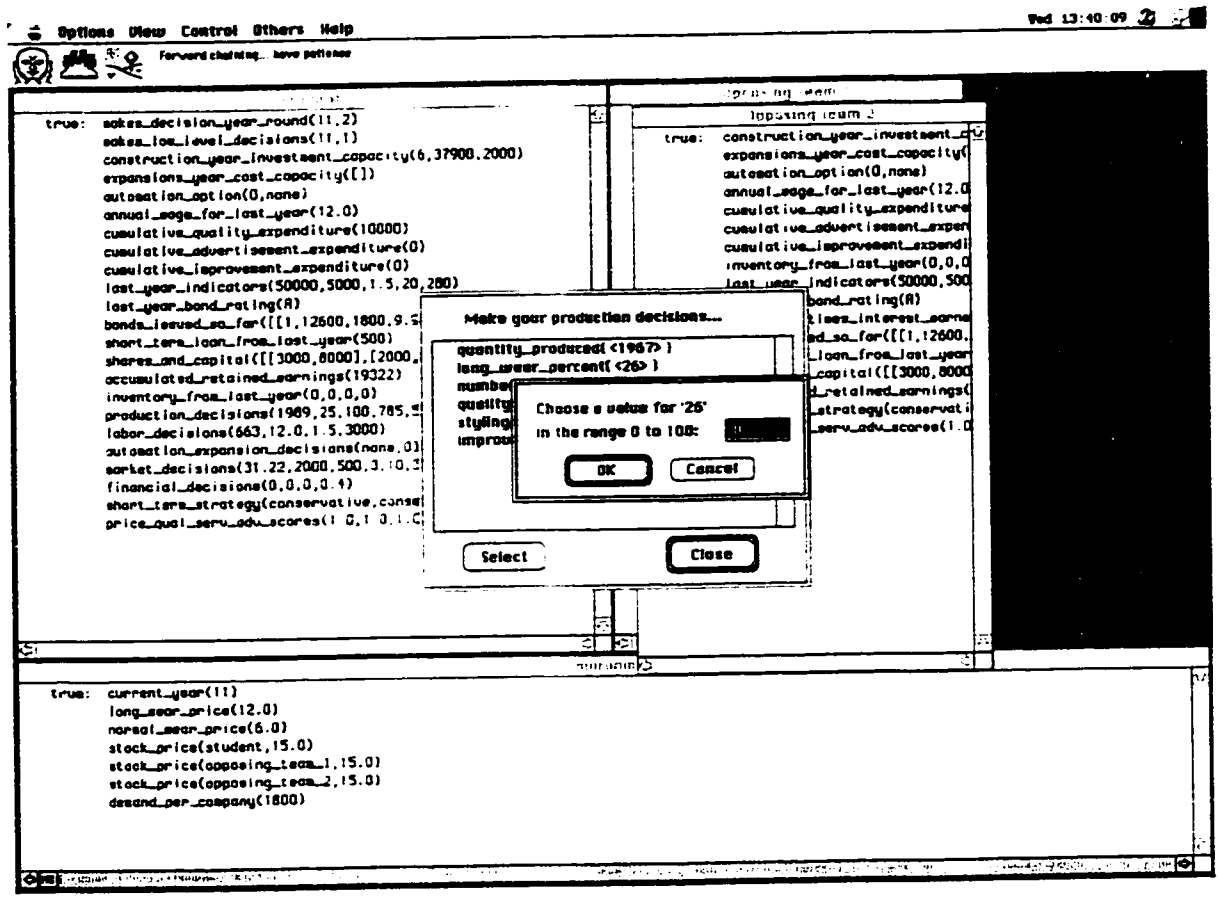

## Appendix D.6. THE LOW-LEVEL DECISION-MAKING MENU

The user selects all the decision categories in order to make updates in the decisions made by converting the high-level decisions. The user also wants to continue with the main selection menu after making decisions for each category.



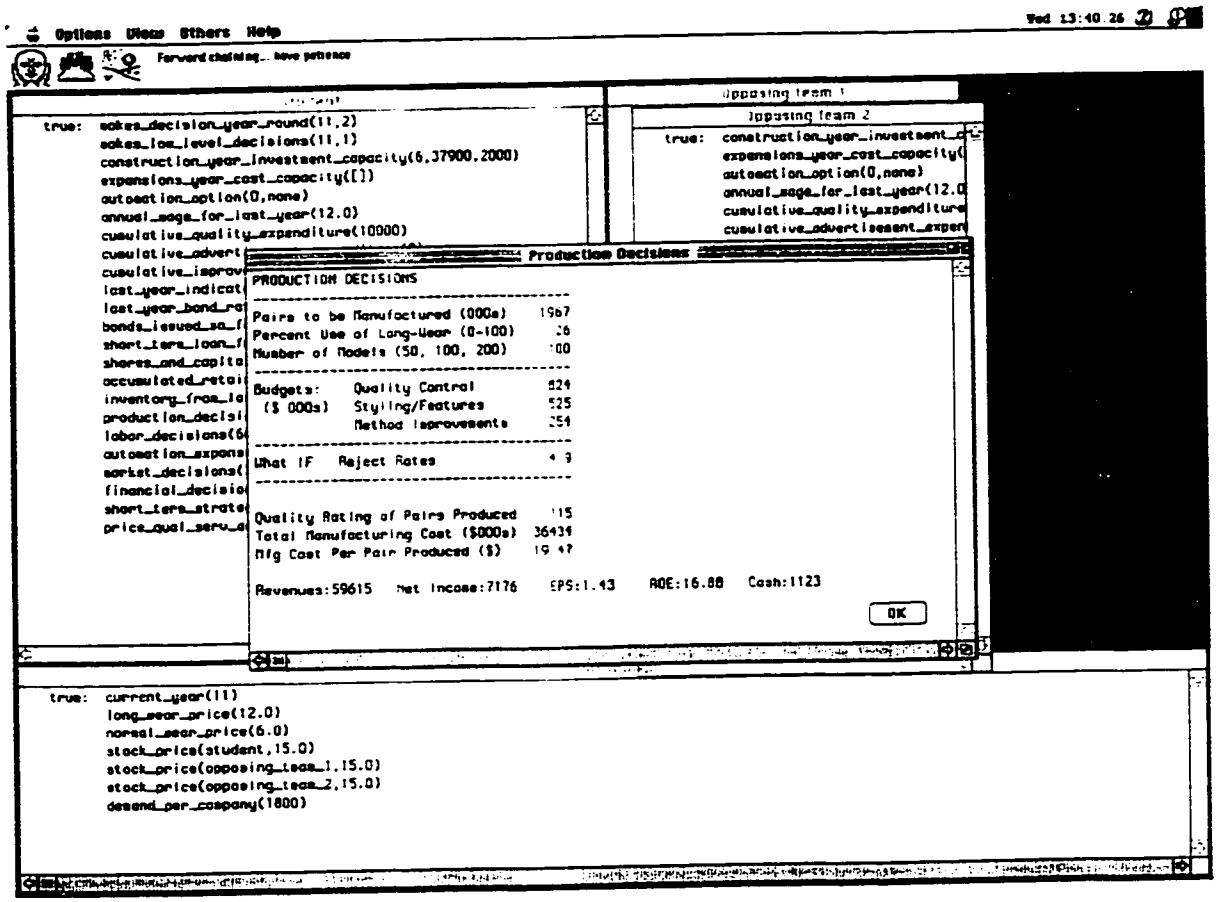

## Appendix D.7. THE QUANTITATIVE PRODUCTION DECISION-MAKING MENU

The previous production decisions appear as the default values in the selection menu.




## Appendix D.8. A MODIFICATION IN THE LONG-WEAR PERCENTAGE DECISION

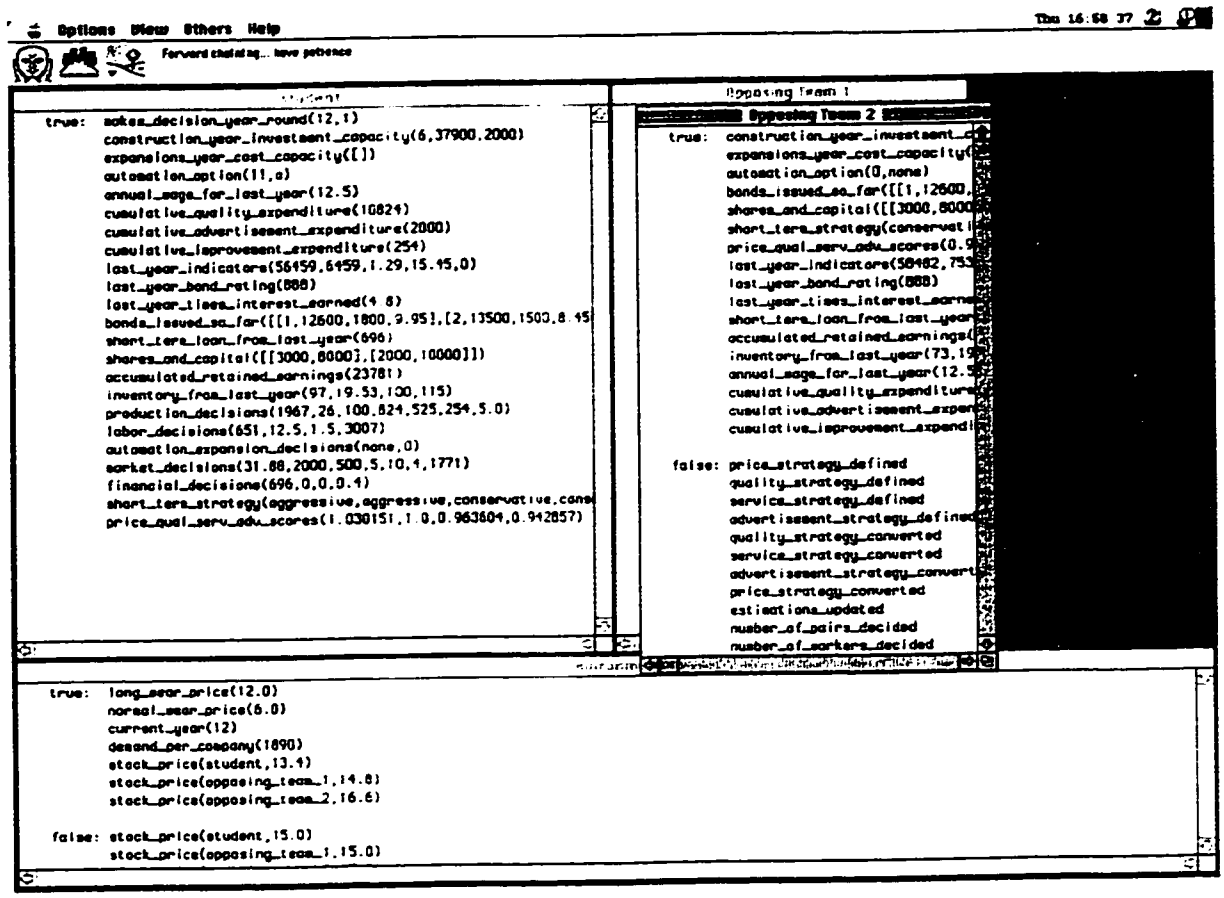
The user wants to make a change in the long-wear percentage decision.




## Appendix D.9. THE PRODUCTION DECISIONS SCREEN

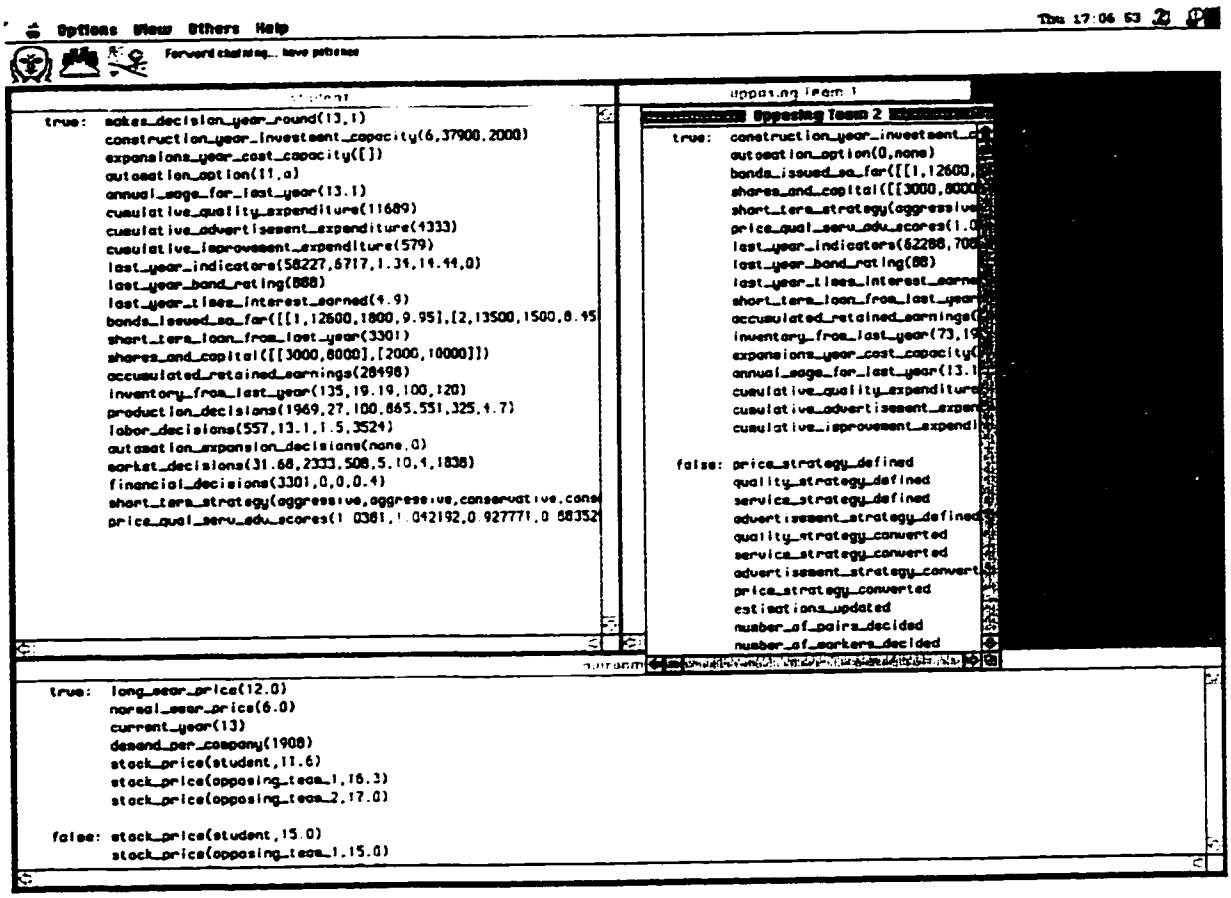
This screen is displayed after the production decisions have been made.

Similar screens are displayed for all the other decision categories, after the decisions in those categories have been modified.



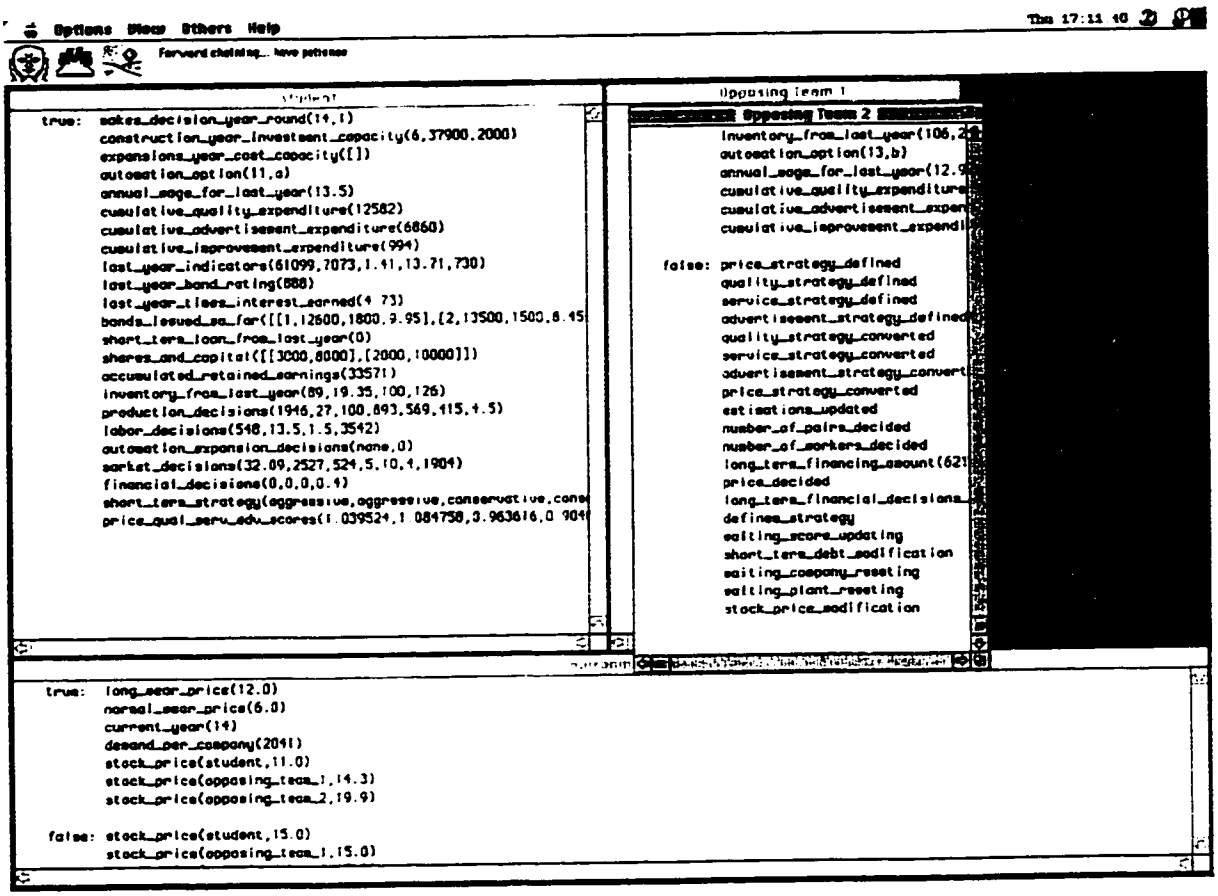

## Appendix D.10. THE RESULTS OF THE FIRST YEAR DECISIONS

This is the state of the companies at the end of their 11th year.




## Appendix D.11. THE RESULTS OF THE SECOND YEAR DECISIONS

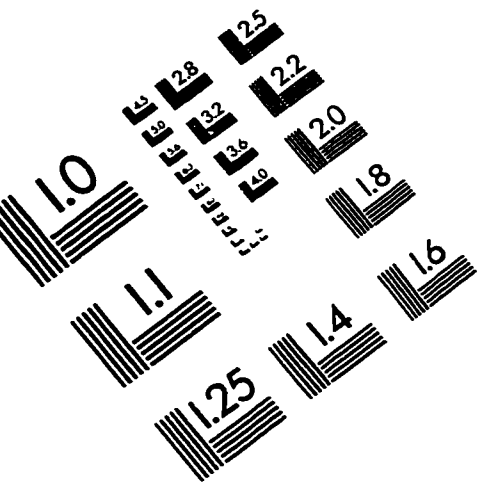
This is the state of the companies at the end of their 12th year.

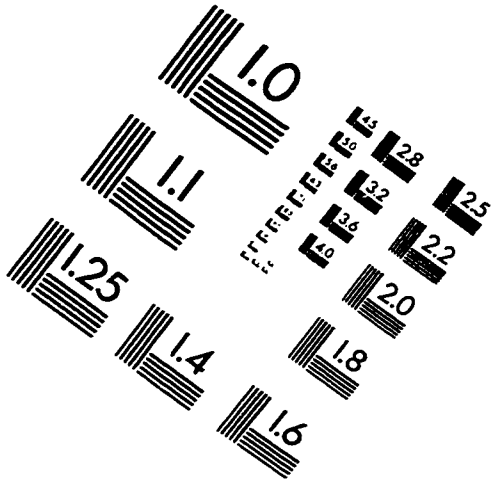


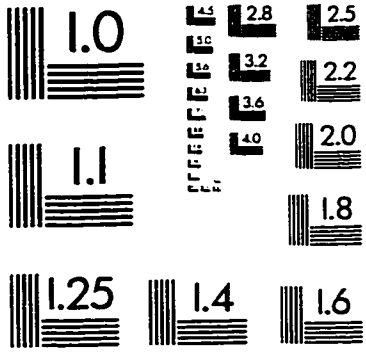

## Appendix D.12. THE RESULTS OF THE THIRD YEAR DECISIONS

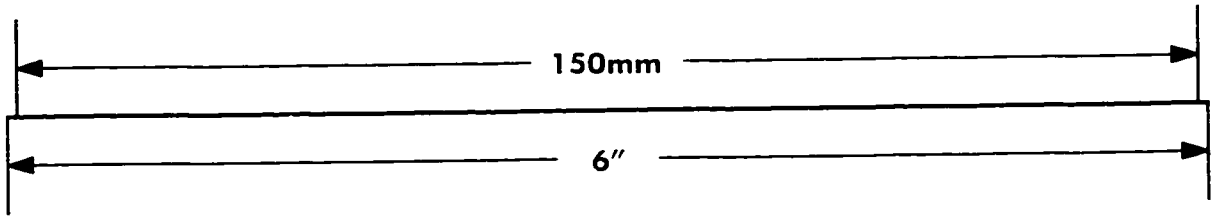
This is the state of the companies at the end of their 13th year.

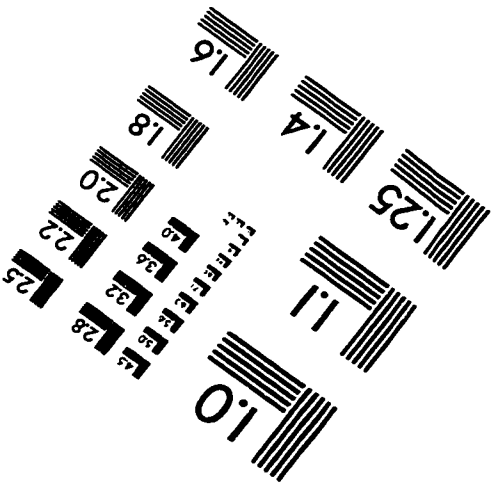
# IMAGE EVALUATION
# TEST TARGET (QA-3)

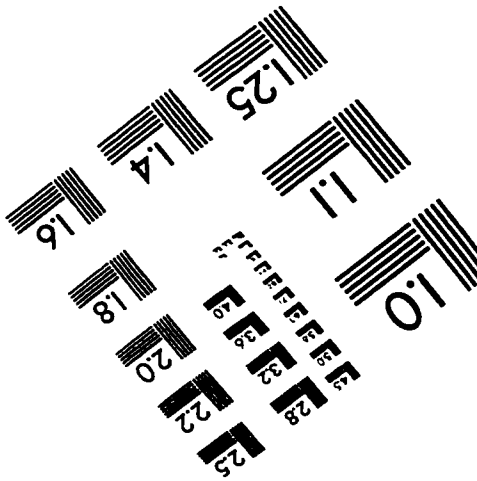



APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved