A MODEL THEORY APPROACH TO PROBLEM SOLVING SYSTEMS DEVELOPMENT

A MODEL THEORY APPROACH TO PROBLEM SOLVING SYSTEMS DEVELOPMENT

Yoshio YANO

March, 2006

Submittd in Partial Fulfillment of the Requriements for the Degree of
DOCTOR OF ENGINEERING

Dissertation Advisor: Yasuhiko TAKAHARA

Department of Management Engineering
Chiba Institute of Technology

Acknowledgments

First of all, I wish to express my utmost gratitude to my supervisor, Professor Yasuhiko Takahara, for his insightful guidance throughout my studies at Chiba Institute of Technology. His advice and direction were indispensable in planning and carrying out the research contained herein.

I am also grateful to Professor Naoki Shiba and Professor Toru Takagi, who helped me for understanding of computer science and mathematics.

In addition, I am highly thankful to Professor Julianne Belbin Sakuma, who helped me for correcting my poor English.

Last, but not least, I also thank my colleagues, who studied together at Takahara Lab., and all other people for their support.

Abstract

This dissertation proposes a new formal approach, which is called a Model Theory Approach, to the design and the implementation of an extSLV (extended solver). The extSLV is a main engine for a problem solving system, which is one component of a Management Information System (MIS). The MIS consists of two subsystems; a transaction (data) processing system and the problem solving system.

The MIS development in the Model Theory Approach is based on a model theoretic structure derived from systems concepts and models of the General Systems Theory. A system developer represents a model, which has the model theoretic structure, by the Set Theory and the Logics, and implements the model in a forth generation programming language, extProlog. The extProlog is an extension of a standard Prolog to allow for the implementation of the MIS.

Traditionally, the MIS has been developed without using formal methods. By the informal methods, the MIS is developed on its lifecycle without having any models. It causes many problems such as lack of the reliability of system design specifications, too late discovery of errors in the system, and so on.

To overcome these problems, some formal approaches to the MIS development have been developed. A VDM-sl (Vienna development method-specification language) is one of the most popular methods. It designs a system based on a system model, and the model is described in the Set Theory and the Logics. Although I have to say that it is not the Set Theory but rather a

new programming language. In addition, it is far from the implementation because it is just a formal specification research.

Our approach is different from the VDM-sl because our approach only uses the pure Set Theory and the Logics, and also makes it possible to implement the executable system right after the modeling.

Additionally, speaking of the problem solving systems development, the focus of every conventional research is on discovery of an algorithm for particular problem. In contrast, our research targets at the consideration for the problem solving system as a system and systems development thereon. We have been trying to develop the simple yet universal approach to the design and the implementation of the problem solving system at the cost of optimum performance of the algorithm.

Moreover, since most of the problem solving systems are executed on workstations and personal computers because of the rapid increase of their computing power and the decrease of their cost, there is a real need for a systems development methodology targeting at small and medium scale systems. In addition, one of the most important things of the problem solving systems development is the understanding of the target problem, and the person who is most familiar with the problem is an end user (EU). Therefore the approach must be EUD-oriented (end user development). The Model Theory Approach is the approach that can meet those requirements.

This dissertation will show the model theoretic structure of the problem solving system, its formulation and implementation. It also demonstrates the availability of the approach by applying it to two simple problems.

Contents

Chapter 1 Introduction

1.1 Objective and Background of the Study

Our group has been proposing a Model Theory Approach that is a new development methodology for a Management Information System (MIS). This new development methodology for the MIS is based on a model theoretic structure [5] derived from the systems concepts of the General Systems Theory [8]. It represents a model in Set Theory and Logics, and implements the model in a fourth-generation programming language, extProlog [2, 4, 1]. The extProlog is a fourth generation programming language, which is an extension of a standard Prolog to allow for the implementation of the MIS [16, 12, 7, 3].

The MIS can be modeled as a hierarchical system consisting of two types: a transaction (data) processing system (TPS) and a problem-solving/decision-support system. Although the Model Theory Approach covers both of two systems, this thesis focuses on the only one component of the problem-solving/decision-support system, an extSLV, and introduces the Model Theory Approach to its design and implementation.

Figure 1 shows the general scheme of the MIS.

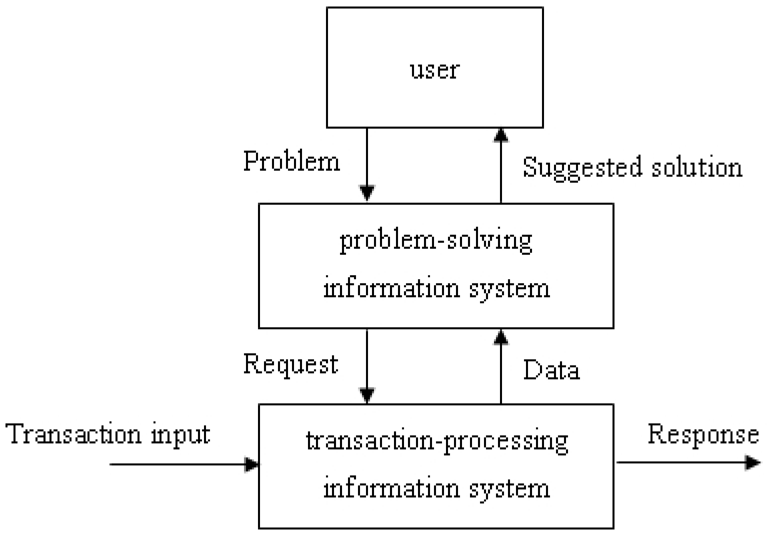

user
Problem
Suggested solution
problem-solving information system
Request
Data
Transaction input
transaction-processing information system
Response

Figure 1. General MIS model

A TPS is an input-output system, with a transaction input and a request from a user as inputs, and a transaction output and a response to the request as outputs. The transaction output is typically a modification of a file system within the TPS.

A problem-solving information system is modeled as a goal seeking system (refer to Section 2.1), in which the problem (assumed to be associated with a goal) is specified by the user and data from the TPS. The function of the goal seeking system is to produce a solution satisfying the goal or to suggest solution candidates to the user.

The TPS traditionally constitutes the infrastructure of the MIS and the information system is understood as being almost equivalent to a transaction processing system. However, the problem-solving/decision-support system has become more and more important as requirement levels for the information system have risen.

The motivation for the new systems development methodology proposed by our group is founded on the following observations:

(1) The rapid increase in computing power and decreasing cost of workstations and personal computers means that most information systems are now developed on these platforms. Decision makers can also solve their problems on these platforms. As such, a new methodology targeting workstations and personal computers must be explored.

(2) Companies are now facing global competition in a rapidly changing business environment. Decision makers need to make the right decisions at the right times, and require technological support in order to do so. However, the conventional systems development methodologies such as the lifecycle approach are notoriously slow and unwieldy, and as such are not suitable for the problem solving function. A light, rapid systems development methodology is therefore required.

(3) Traditionally, systems development follows the way in which a user puts forward the request for information, and the system developer then develops an appropriate system. However, as an image of the user is semi-structured or unstructured and also changeable, it is very difficult for the system developer to understand the request of the user precisely. Furthermore, the users themselves may only know a certain condition of the problem, without understanding the true nature of the problem from the beginning, and quite often a significant amount of time is spent interacting with the solution process before obtaining a clearer perspective of the problem. It is therefore the user who is best equipped to build a system, indicating that a methodology oriented around end-user development (EUD) is necessary.

(4) Almost every nontrivial system is built assuming that it is to be modified during operation. In general, however, the end user must have detailed knowledge of the system or prior involvement in its development in order to modify the system effectively. This issue may be resolved by the introduction of a fast and easy systems development methodology.

Figure 2 shows the differences between the Model Theory Approach and other well-known approaches.

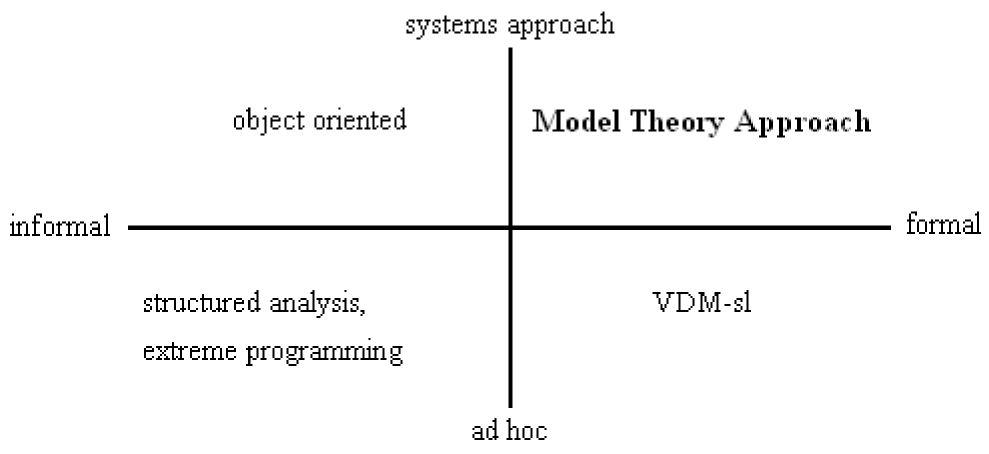


Figure 2. Comparison with other approaches

It is clear that the structured analysis, the extreme programming and the object-oriented approach are not formal approaches. Although the VDM-sl (Vienna Development Method-specification language) [6] is a formal approach, it does not include a system model for an information system such as the Model Theory Approach. The VDM-sl is one of the most notable examples. Although, it relies on the Set Theory and the Logics, its language had better be understood as a computer language with strict type requirements rather than the Set Theory. It is certainly desirable for an ordinary system developer not to learn another involved computer language in addition to the Set Theory, whereas study of the Set Theory we believe is the minimum requirement for any professional system engineer. Another problem with the VDM-sl is that as the name indicates, it is principally concerned with the specification phase of the system development. Advantage of formalism cannot be fully appreciated if it is relevant only to specification.

The Model Theory Approach is different from the VDM-sl. In general, the Model Theory Approach provides an integrated platform on which both the problem solving systems development and the TPS development can be dealt with. The Model Theory Approach is based on systems model of the TPS and the extSLV, and uses the Set Theory in the standard form. It also includes an implementation phase of the systems development as its integral component.

Many papers written by our group cover the parts of the Model Theory Approach where this dissertation does not deal with. Reference 18 and 21 discusses the approach to the TPS. The extSLVs for many example problems are developed in the Reference 17, 19 and 20.

This dissertation will propose the new formal approach to the development of the extSLV. It will reveal the structure of the extSLV, and will show its formulation and implementation. As a result of them, the dissertation will provide a theoretical basis for the extSLV development, and will realize a rapid systems development.

1.2 Target of the Study

The target of the Model Theory Approach is the entire Management Information System (MIS) illustrated in Figure 1. Although its structure is too general to produce meaningful results, it can be specified according to the objectives of the proposed Model Theory Approach. Figure 3 shows a model of an "intelligent" MIS for the support of the intelligent business activities, representing the ultimate target of the Model Theory Approach.

In Figure 3, the TPS of Figure 1 is represented as an automaton model in which the database (DB) saves the current state while the date processing system (DPS) plays the role of the state transition function [14]. The transaction input into the DPS is from the user, and the response from the DPS is passed to the user.

The problem-solving information system of Figure 1 is decomposed into two levels in Figure 3: the problem solving system and the data transformation system. The data transformation system transforms the data of the TPS into information by the abstraction (aggregation) and/or the data mining [20].

Although the TPS is a principal target of the current systems engineering, the problem solving system is much more complicated and theoretically challenging.

The problem solving system is broken down into further two layers: a problem solving layer and an adaptive layer. The two layers and the user (layer) correspond to the control layer, the adaptive control layer, and the self-organization layer of the layer model of the General Systems Theory (GST), respectively [8]. The three-layer model is called a skeleton model [13].

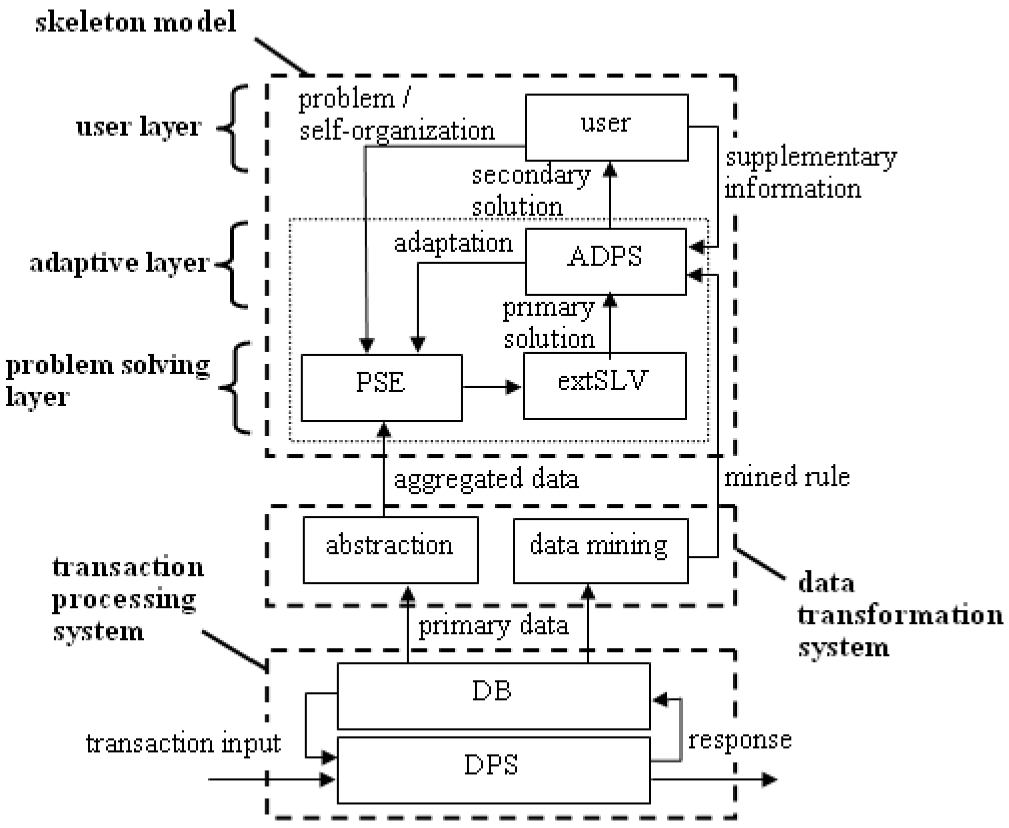


Figure 3. Intelligent Management Information System

The problem solving layer, the lowest layer of the skeleton model, usually consists of two components: the problem specification environment (PSE) and the extended solver (extSLV), as shown in Figure 3. The PSE specifies the structure of the problem to be solved, but is in general not a problem representation. The problem is instead formed and solved in the extSLV. As the problem is usually specified outside of the solver, the solver in Figure 3 is called an extended solver, the output of which is the (primary) solution. The user inputs into the PSE not only a problem but also a self-organization, which is the input that changes the structure of the problem. Besides, there is an input from the user into the ADPS, supplementary information. The ADPS deals with the uncertainty from the external world, and the supplementary information from the user is to control the ADPS.

The goal is to establish a general basis for construction of a problem solving system, specifically the design and implementation of the extSLV using a formal structure of the problem solving and the extProlog. It should be noted that because the problem solving layer is responsible for the primary activity of the problem solving system, the extSLV is a real engine of the problem solving system.

The principal function of the extSLV is to yield an optimized (or satisfactory) solution for an identified problem. Although optimization may appear to be too special for the MIS concept, it should be understood here in the context of the intelligent MIS model. The adaptive layer and the user layer deal with uncertainties associated with a problem.

Although the target of the Model Theory Approach is all parts of the MIS illustrated in Figure 3, this dissertation only focuses on the development of the extSLV.

1.3 Chapters of the Dissertation

This dissertation is composed of six chapters including Introduction.

Chapter one, “Introduction”, restricts the objective of this dissertation, and defines the target discussed in this dissertation. It also shows its preliminaries and background.

Chapter two, “extSLV Model Based on Mathematical General Systems Theory”, Chapter three, “User Model”, and Chapter four, “Standardized Goal Seeker”, are the main part of the dissertation, which propose the Model Theory Approach for the extSLV. Chapter two first gives the system models and concepts of the Mathematical General Systems Theory which are applied to the extSLV. The second section of the chapter shows the extSLV model generated by applying them. Additionally the development procedure of the extSLV in the Model Theory Approach is briefly discussed. Then the last section shows the classification of problems whose extSLV is developed by the Model Theory Approach.

The extSLV model is decomposed into two parts. One is called a user model; the other is called a standardized goal seeker. Chapter three and four discuss the user model and the standardized goal seeker, respectively.

Chapter five uses two examples, WGC Problem and Traveling Salesman Problem to show how their extSLV can be generated by the Model Theory Approach. Then Chapter six is the conclusion.

Chapter 2 extSLV Model Based on Mathematical General Systems Theory

This chapter will discuss an extSLV model that we developed by applying system models and concepts of the Mathematical General Systems Theory. In the Model Theory Approach, the extSLV model that I will discuss in this chapter can be the basis for extSLV design. In other words, it can be a general template that can be applied to the development of solvers for any problems.

This chapter will start by showing the system models and concepts, which were applied to the development of the extSLV model. At the beginning of the first section, I will establish what the problem and the problem solving are, then show a problem solving model of the General Systems Theory. Then I will show the developed extSLV model. In addition, this chapter will describe the development procedure of extSLVs in our approach, and also describe the classification dimensions of problems that should be useful for the extSLV design.

2.1 System Models and Concepts for extSLV

In this section, I describe the applied system model and concept of the Mathematical General Systems Theory to the development of an extSLV model. Before that, I establish the problem and the problem solving.

Figure 4 is a general concept of a problem and a problem solving that the Model Theory Approach adopts.

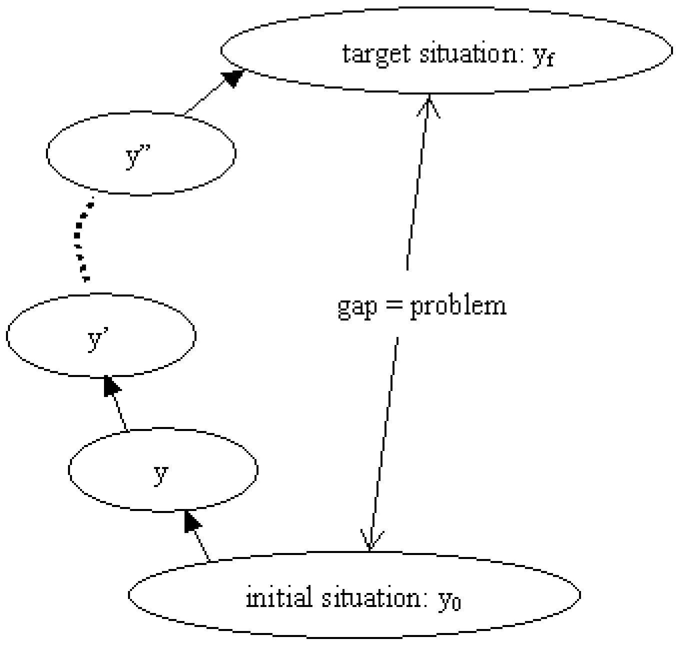


Figure 4. General concept of problem and problem solving

As the figure shows, a problem is defined as a gap between an initial situation, y_0, and a target situation, y_f. Then the problem solving can be defined as attempts to arrive at the target situation, y_f. In fact, we consider that each one of the situations, y_i, indicates a state of the problem solving, and the state goes close to the target state, y_f, by applying an appropriate action, a, to a current state, y_i.

There is one thing that we should consider about the solution of the problem, given this perspective. That is, just because we arrive at the target state, does not mean it is the solution. In some cases of problems, it is required to reach the target state effectively.

As the model that realizes the problem solving activity of Figure 4, we use a goal-seeking model of General Systems Theory illustrated in Figure 5 [8]. The model is specialized to suit the purpose of the Model Theory Approach.

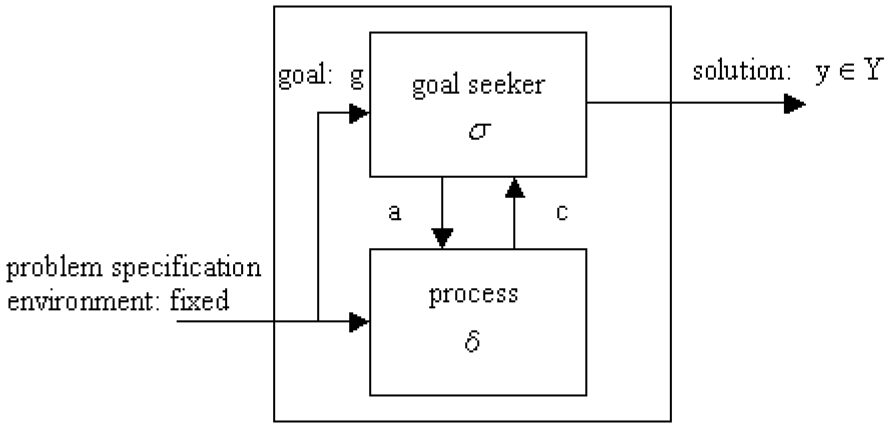


Figure 5. Goal-seeking model

As the figure shows, the extSLV is an input-output system whose input is a problem specification environment and whose output is a solution. The model consists of two components, a process and a goal seeker.

The process is an abstract model of the problem solving activity of Figure 4, which is the expression of the transition process of the state. The process has two inputs; external one and internal one. The external input specifies the target problem that is the problem specification environment. We consider that the problem specification environment specifies the structure of the process and is supposed to be fixed during the problem solving activity. On the other hand, the internal one is a parameter, *a*, that is a control variable or a decision action. The output of the process is a state of the solving activity.

In the meantime, the goal seeker specifies a parameter, *a*, to achieve the given goal, *g*, using the state information, *c*. It yields a solution, *y*. The goal, *g*, is the function that gives a criterion to determine which action should be selected for each state.

It might be obviously given one legitimate goal for a problem, but it often happens that a legitimate goal cannot be found and we should give it in the heuristic way. The Traveling Salesman Problem is the problem that has a legitimate goal because if a traveling route, which is the output, is given, its evaluation measure, the traveling distance, is determined.

The output is obviously the solution. It could be a string of the actions such as *aa'...a"*, and could also be the result generated by the string of the actions.

The extSLV model that will be shown in the next section was realized by applying the goal seeking system.

2.2 extSLV Model

The extSLV model implemented by applying the above system model and concept is illustrated in Figure 6. The different part from Figure 5 is the goal seeker part. As I mentioned above, the goal seeker determines the appropriate action, *a*, for each state, *c*. To do this, it must have a strategy. As the result of our research, we found a useful strategy that is called the hill-climbing method with a push down stack (hereinafter called PD method). Figure 6 illustrates the extSLV model with the PD method.

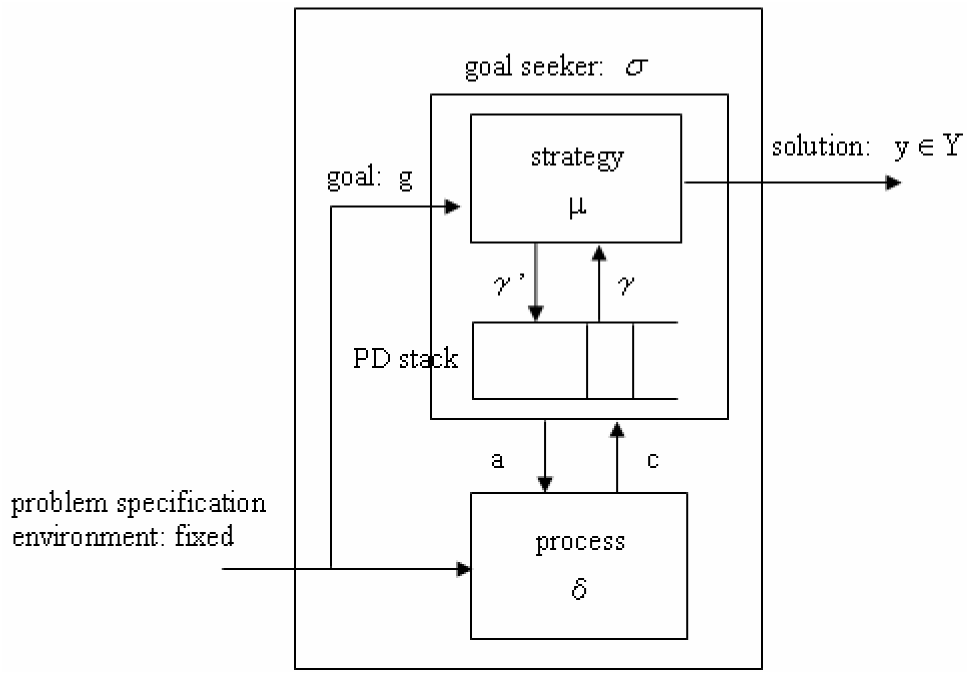


Figure 6. extSLV model with hill climbing method with push down stack

It has two components, a strategy, μ, and a push down (PD) stack. As Figure 4 shows, the current state goes close to the target state by applying a selected action to the current state. This activity can be called a hill climbing [10]. During the hill climbing activity, the system often falls into a deadlock situation. It is the situation that the system meets the state that has no available action. Figure 7 illustrates the situation by tree search.

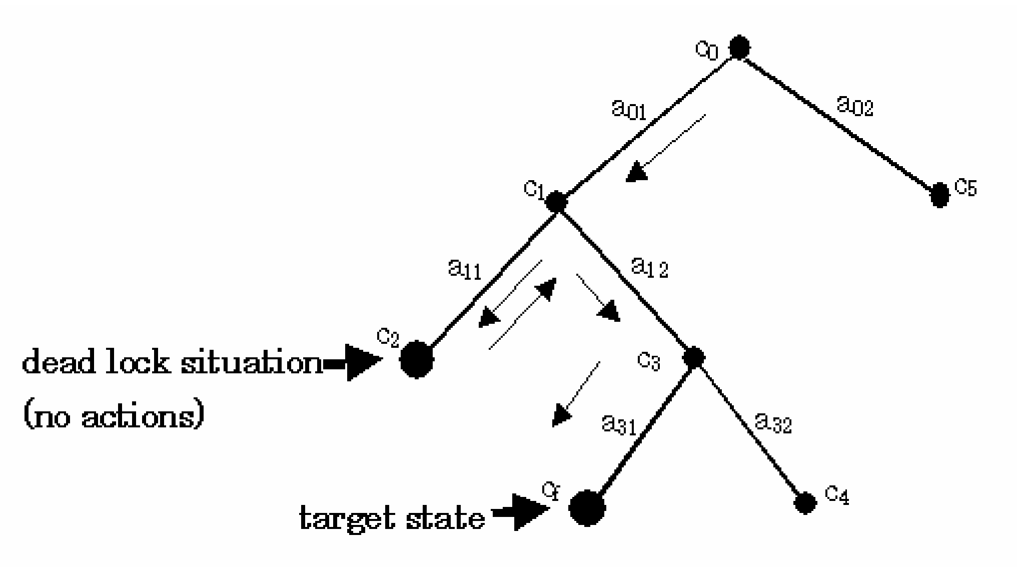


Figure 7. Deadlock situation

If the goal, g, chooses a left-hand action, a_{11}, at the state, c_1, the system arrives at a state, c_2. However, there is no action at the state c_2, so the system cannot proceed anymore. This is the deadlock situation.

When the system meets this situation, it must make it possible to go back to a previous state such as a state, c_1, for the state, c_2, and try another action. In order to make it possible to do so, the push down stack is provided, and stores the information for the backtracking. One piece of the information has four elements; c_i, *RA*, *Sol2* and *H2*. c_i is the current state. *RA* is a set of remained actions at the state, c_i. *Sol2* and *H2* are a string of selected actions and a string of states until c_i, respectively. The push down stack can be written in the top and can be read from only the top.

The formal discussion of the PD method is not made in this section, but it will be treated in Section 4.1. As the result of the modeling of the extSLV, we found that the extSLV model could be decomposed into two parts, and one could be standardized. It is the goal seeker part.

The other part cannot be standardized and it depends on the target problem, it must be designed by system designers. It is a combination of the process part and the goal, g, and it is called a user model. The design of the goal seeker is the most difficult part of the extSLV development, so its standardization can be very helpful for system developers.

2.3 Development Procedure of extSLV

Figure 8 shows the development procedure of the extSLV as illustrated in Figure 6.

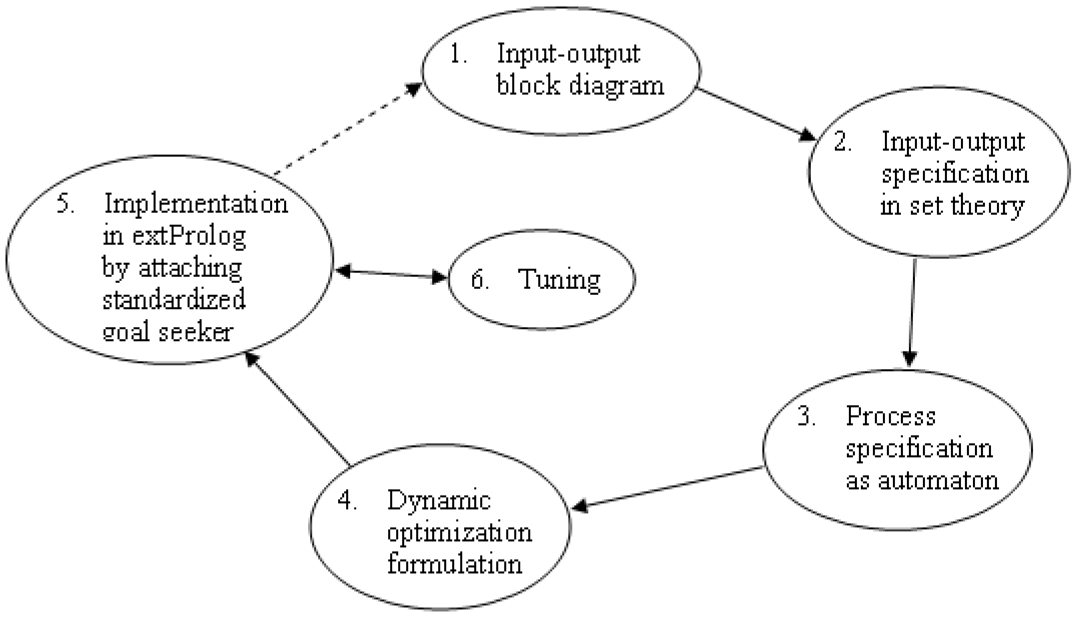

1. Input-output block diagram
2. Input-output specification in set theory
3. Process specification as automaton
4. Dynamic optimization formulation
5. Implementation in extProlog by attaching standardized goal seeker
6. Tuning

Figure 8. Design and implementation procedure

The development procedure consists of six stages: input-output block diagram, input-output specification in Set Theory, process specification as automaton, dynamic optimization formulation, implementation in extProlog by attaching standardized goal seeker, and tuning.

As I mentioned above, the extSLV model is divided into two independent components of the user model and the standardized goal seeker. As the user model represents the dynamism of a problem solving activity as appropriate for the target problem, it must be specified as a user model by a system developer following the implementation structure of the Model Theory Approach, whereas the goal seeker is given as a standardized component prepared for the formalized structure of the solving activity. The minimum task required for the developer are then (i) formalizing the problem solving activity in the automaton form, and (ii) calling the goal seeker to manipulate the automaton [9].

Although the standardized goal seeker can perform its function satisfactorily for many problems, it may not be efficient for certain problems. In such cases, tuning must be performed for the goal seeker after a working extSLV has been obtained.

The most difficult part of automaton formulation may be the identification of the state. As shown in Chapter 3, however, the present methodology provides a basic and general procedure for identifying the states for a given problem.

2.4 Classification of Problems

It is clear that the implementation of the design procedure depends on the character of the problem under consideration. In order to make the procedure operational, a classification of problems is required.

In general, problems can be categorized into those that are suitable for a conventional solver algorithm and those that are not [11]. In some fortunate situations, the PSE can be formulated as a problem to which a conventional solver (or solving algorithm) is directly applicable. Figure 9 outlines this case, in which the extSLV is decomposed into a problem formulator (PRF) and a conventional solver.

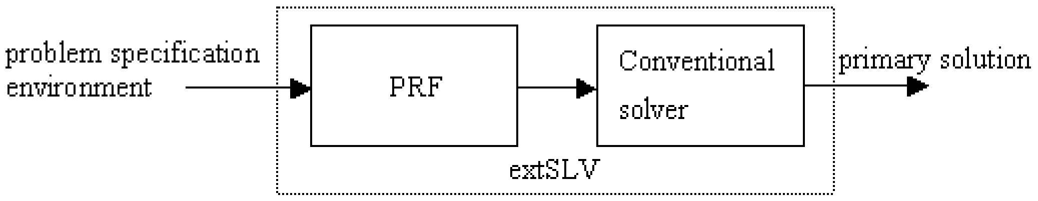


Figure 9. Conventional extSLV

A typical example of a conventional solver is the linear programming (LP) algorithm, which requires a problem to be represented in a standardized form. The PRF in Figure 9 transforms the parameters of the PSE into a standard form as required by the solver. In this case, problem identification may be difficult because the problem must be formulated to satisfy the required form. The treatment in this thesis is concerned with the latter case, which is more common in practices. The latter case requires further classification of the problem.

The Model Theory Approach uses the following three dimensions for the categorization of problems in cases when a conventional solver is not applicable:

1. Explicit solving action – implicit solving action (E/I).
2. Closed goal – open goal (C/O).
3. Closed target – open target (C/O).

An explicit solving action indicates that an output (solution) of the extSLV is a sequence of actions of the solution process illustrated in Figure 4. The internal dynamic process, δ, in Figure 6 is then defined by the PSE and the output in a natural way, as will be shown in Chapter 3. Typically, the traveling salesman problem, which is discussed in Chapter 5, has an explicit solving action, whereas the knapsack problem does not. The derivation of a specific solving activity is not relevant to the solution of the knapsack problem.

A closed goal problem implies that evaluation of the output is given by the problem specification environment. The traveling salesman problem is a closed goal problem because if a traveling route (output) is given, its evaluation measure, the traveling distance, naturally follows.

A closed target problem is a problem for which the final state (c_f in Figure 4) of the solving process can be directly determined when process, δ, is specified. In general, the final state is not uniquely specified. The traveling salesman problem can be described as a closed target, while the knapsack problem cannot. If the problem is an open target, the stopping condition of the solving activity cannot be easily determined. It should be noted that even if the goal seeker reaches a final state, it does not necessarily mean that a solution is obtained. A path to a final state may be a solution only if it satisfies a specific criterion, for instance, to be the shortest.

There are eight possible cases for classification. Table I lists the classifications and examples.

Table I. Problem classifications

E-C-C problem	Traveling salesman problem
E-C-O problem	Linear quadratic dynamic optimization problem
E-O-C problem	WGC problem
E-O-O problem	General control problem
I-C-C problem	Cubic root problem
I-C-O problem	Knapsack problem
I-O-C problem	Class scheduling problem
I-O-O problem	Data mining problem

The traveling salesman problem and the knapsack problem are well known as typical OR problems. The class schedule problem is a problem to develop a class schedule of a school.

Chapter 3 User Model

As I mentioned in the previous chapter, the extSLV model was decomposed into two parts; the user model and the standardized goal seeker. This chapter discusses the user model by following the development procedure of Figure 8.

There exist eight categories of user models according to the classification of Section 2.4. This chapter investigates two extreme cases: the E-C-C case and the I-O-O case. The other cases can be considered as combinations of these two extreme cases.

3.1 User Model for E-C-C Case

This section discusses the user model for the case of E-C-C, explicit solving action, closed goal, and closed target states.

a. Input-output Specification

This section is about stage 1 and 2 of the design and implementation procedure in Figure 8. At the beginning of the extSLV design in the Model Theory Approach, we describe the extSLV as an input-output block diagram. We can say that the extSLV illustrated in Figure 6 can be considered as an input-output system whose input is the problem specification environment and output is the solution or solution candidates of a target extSLV as Figure 10 shows.

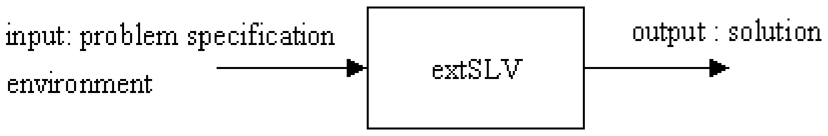


Figure 10. Input-output block diagram for target extSLV

At the first stage of the development procedure of Figure 8, Input-output block diagram, the problem specification environment and the solution are conceptually given, and then they are specified in elementary Set Theory and Logics at the second stage, input-output specification in Set Theory.

The input, the problem specification environment, is generally complicated so it must not be represented as a simple set but as a structure (of the Model Theory [5]). On the other hand, the output, or solution, is generally represented as a set of solution candidates.

The problem specification environment has the following structure:

<class of base sets, class of relations, class of functions, class of constants>.

On the other hand, the output must be represented as a set, Y. The representation of the output set, Y, has some difference depending on the first dimension of the classification of problems, explicit solving action - implicit solving action discussed in Section 2.4.

In the case of explicit solving action, suppose that A is the action set, the output set, Y, is

$$Y \subset A^*,$$

where A is the set of solving actions. A^* is the free monoid of A. An element, $a \in A$, is not usually a simple element but may have a complicated structure, which is determined by the problem specification environment.

This stage becomes both easy and difficult depending on each problem. In addition, it is not completed at once, but is improved though the implementation in extProlog, which is illustrated by a dotted arrow in Figure 8.

b. Process Specification as Automaton

At the third stage, process specification as automaton, the process, δ, of the goal seeking system of Figure 6 is represented as an automaton. At the threshold of the process specification as automaton, it is required to define a state set and an action set depending on the first dimension of the classification of problems, explicit solving action - implicit solving action, discussed in Section 2.4.

In the case of the explicit solving action, since the output set, Y, is given as

$$Y \subset A^*,$$

a general form of δ can be specified as below:

$$\delta : Y \times A \to Y$$

such that

$$\delta(y, a) = y \cdot a$$

where $y \cdot a$ is the concatenation of y and a. Y and A are the state set and the action (input) set of the state transition function, δ, respectively. The output function,

$$\lambda : Y \times A \to A,$$

takes a special form for the current case.

$$\lambda(y, a) = a.$$

In many cases, Y is too large to be used as a state set. Fortunately, usually we can find a set, C, and an onto map, $\omega : Y \to C$, such that ω satisfies the following relation:

$$(\forall y, y', a)(\omega(y) = \omega(y') \to \omega(y \cdot a) = \omega(y' \cdot a)).$$

The function ω is called a state reduction map. It satisfies the following property:

Proposition 3. 1 $\ker \omega \subset \mathrm{Y} \times \mathrm{Y}$ is Nerode equivalence, or ω is an automaton homomorphism [8]. That is, there is an automaton,

$$<A, C', \delta', \lambda'>,$$

such that the following diagrams are commutative:

$$\begin{tikzcd} Y \times A \arrow[r, "\delta"] \arrow[d, "\omega \downarrow I"'] & Y \arrow[d, "\omega"] \\ C' \times A \arrow[r, "\delta'"] & C' \end{tikzcd} \qquad \begin{tikzcd} Y \times A \arrow[r, "\lambda"] \arrow[d, "\omega \downarrow I"'] & A \arrow[d, "I"] \\ C' \times A \arrow[r, "\lambda'"] & A \end{tikzcd}$$

Proof Let

$C'=(Y/\ker\omega)$.

The function, ω, is identified as

$\omega(y)=[y]$.

Let

$\delta':C'\times A\rightarrow C'$

where δ' can be defined as

$\delta'([y], a)=[\delta(y, a)]$.

As a matter of fact, the above definition is valid. If $[y]=[y']$ or $\omega(y)=\omega(y')$,

$\omega(\delta(y, a))= \omega(y\cdot a) =\omega(y'\cdot a)=\omega(\delta(y', a))$

for any a. That is,

$[\delta(y, a)]=[\delta(y', a)]$.

Then, the left diagram is commutative. Similarly, let $\lambda':C'\times A\rightarrow A$ be

$\lambda'([y],a)=a$.

Then, the right diagram is also commutative. Then, ω is an automaton homomorphism from <δ, λ> to <δ', λ'>.

Q.E.D.

If ω is found, $(Y/\ker\omega)$ will be used as a state set. $(Y/\ker\omega)$ is expected to be of a manageable size. If we allow the trivial case that $\omega(y)=y$, ω is always exists. As such the state set will be denoted by C and the state transition function and the output function will be denoted by

$$\delta : C \times A \rightarrow C$$

and

$$\lambda : C \times A \rightarrow A,$$

respectively.

There may be misunderstanding that automaton formulation can be introduced because the target problem has the explicit solving action property or it is a dynamic problem. This is not true. The automaton formulation comes from the dynamics of the solving process.

Generally, the function, δ, is generally a partial function. In order to assure that δ behaves properly, two functions bellow are used:

$$genA : C \rightarrow \wp(A)$$

and

$$constraint : C \rightarrow \{true, false\},$$

where $\wp(A)$ is the power set of A. The function, *genA*, which is called an allowable activity function, specifies allowable or preferable actions for a given state. For instance, because every movement of a token is not allowed at a corner for the 8-puzzle problem, the function, *genA*, is used to ensure the restriction is satisfied. Sometimes *genA* is used to artificially restrict selection of activities using heuristics about the target problem. That is, *genA* specifies a set of preferable activities. The predicate, *constraint*, determines whether a state is permissible. Although any jewel can be put into the knapsack for the knapsack problem, the result state may be prohibited due to the problem restriction. The predicate, *constraint*, is used to assure the restriction. The predicates, *genA* and *constraint*, are effective means to represent heuristics about a target problem. The function, δ, is naturally assumed to satisfy

$$\delta(c, a)=c' \rightarrow a \in genA(c) \text{ and } constraint(c')=true.$$

In addition, let the initial state be

$$c_0 \in C$$

where $c_0 = \omega(\Lambda)$ for the null string, $\Lambda \in A^*$. Obviously,

$$constraint(c_0)=true$$

should hold.

Then, we give a target state. It gives a stopping condition of problem solving process of Figure 4. To do this, we must consider the second dimension of the classification of problems, Closed target - open target.

In the case of Closed target, there is no problem. For many problems, there exist multiple target states, hence we gives a set of target states

$$C_f \subset Y(\text{or } C).$$

C_f may be or may not be a singleton set.

$$(\forall c_f \in C_f)(constraint(c_f)=True)$$

must hold.

The stopping condition for the solving activity is st:C→{true, false} where

$$st(c)=true \leftrightarrow c \in C_f.$$

An automaton specification of the process of the case of explicit solving action is captured by the following tuple:

$$< A, C, \delta, \lambda, genA, constraint, st, c_0, C_f >.$$

The next stage is Dynamic Optimization Formulation.

c. Dynamic Optimization Formulation

When the automaton formulation of the solving process is given, the goal seeker selects an appropriate action, *a*, from *genA(c)* for a given state, *c*. It is made to the function, $goal{:}C \rightarrow Re$, takes the most desirable value. *Re* is a set of reals. Since a state transition function is a discrete state space representation, if an evaluation function, $goal{:}C \rightarrow Re$ is introduced, we have a user model representation which is provided as a dynamic optimization problem.

We assume that *y* is desirable if *goal(c)* is small.

In the case of closed goal, which is the third of the problem classification,

$$goal{:}C \rightarrow Re$$

is given from the problem. The Traveling Salesman Problem is the typical example.

In this way, if the evaluation function, *goal*, is given, the user model is formulated as a dynamic optimization problem as below:

$$user\ model = <A, C, \delta, \lambda, genA, constraint, goal, c_0, C_f>.$$

This problem is to optimize the function, *goal*, assuming *A* as the set of variables.

An extSLV component is a combination of the user model and a goal seeker. The next chapter deals with the goal seeker which is provided as the standardized one.

3.2 User Model for I-O-O Case

This section discusses the user model for another extreme case of I-O-O, implicit solving action, open goal, and open target states [15].

a. Input-output Specification

This stage is almost the same as that of the previous section. The only difference is the existence of implicit solving activity. The output (solution) is not given in a relation with a solving activity. That means we cannot define the output set, *Y*, as a set of strings of the action set. $y \in Y$ can have an involved structure reflecting the property of the target problem.

b. Process Specification as Automaton

At this stage, the solving activity process, δ, is determined as an automaton. Because the problem is of implicit solving activity, the state transition is specified in a way different from that in Section 3.1. In general, let

$$A=\{a \mid a:Y \to Y\}.$$

Then, δ:Y$\times$A$\to$Y is defined as

$$\delta(y, a) = a(y).$$

It is obvious that *A* is often too large to be manipulated conveniently. In order to overcome this difficulty, heuristics can be used to reduce the size of A [6]. This thesis assumes that A is represented as a parameter set rather than as a set of mappings.

We use the symbol *C* for the state set, i.e. δ is represented as $\delta:C \times A \to C$. In general, δ is also a partial function in the current case. Then, the following two functions are used to ensure that δ behaves properly:

$$genA:C \to \wp(A)$$

and

$constraint : Y \rightarrow \{true, false\}$.

genA can be used to effectively reduce the action set depending on the state. δ must satisfy

$\delta(y, a)=y' \rightarrow a \in genA(y)$ *and* $constraint(y')=true$.

The output function, $\lambda : C \times A \rightarrow Y$, is different from that of Section 3.1. In the current case, it is defined as

$\lambda(y, a)= y$.

Let the initial state be

$c_0 \in C$.

Usually, a trivial state is used to specify c_0. If an element of C is a set, $c_0=\phi$ (empty set) is a possible candidate. Obviously,

$constraint(c_0)=true$

must hold.

In the case of open target, the set of target states, C_f is not given a priori. We determine the target in the following three ways:

– Heuristically decide the desirable state, and then consider it as the target state.

– Consider it the target state when there is no possible action.

– Not assume any target state.

The knapsack problem is one example of the first way. However the knapsack problem is the problem to find a optimal condition, it is not known generally.

We use the second way for data mining to generate some rules, for instance [19]. Its target state is not known, but we assume the target state as the state when it becomes impossible to generate any new rules.

In the third way, we determine the stopping condition in the situation of goal discussed below. As noted above, the target state is essential to determine the stopping condition.

An automaton specification of the process of the case of implicit solving action is captured by the following:

$$< A, Y, \delta, \lambda, genA, constraint, c_0, C_f >.$$

The next stage is Dynamic Optimization Formulation.

c. Dynamic Optimization Formulation

Because the current problem is of the I-O-O type, there is no legitimate goal for the problem. Nevertheless, a goal is needed and is used to evaluate an action even in the case of open goals. Heuristics are again used for determination of a goal. In many cases, an estimation of the length of the solving path is used, or desirability of an output is described using a conventional form (for example, a quadratic form). These heuristics will be illustrated in Chapter 5. Let a goal be represented as

$$goal: C \rightarrow Re.$$

If $C=Y$ has a preference relation, $\leq$, it is desirable that the following compatibility relation holds:

$$c \leq c' \rightarrow goal(c) \geq goal(c').$$

Naturally, this relation cannot always be satisfied in practice.

A dynamic optimization formulation of the process is then given as below:

$$<A, C, Y, \delta, \lambda, goal, genA, constraint, st, c_0, C_f>$$

where st:C→{true, false} is a stopping condition, i.e.,

$$st(\hat{c})=true \leftrightarrow \hat{c} \in C_f \,\&\, min\{goal(c) | c \in C_f\}=goal(\hat{c}).$$

Chapter 4 Standardized Goal Seeker

An extSLV component is a combination of the user model described in Chapter 3 and a goal seeker. This section discusses the theoretical basis of the goal seeker.

The goal seeker developed for the structure of the user model can be applicable to any problems if they are formalized into the structure. Therefore, we call it a standardized goal seeker. This fact shows a decomposition theory for the extSLV because it is a combination of a formalized problem and a standardized goal seeker.

4.1 Hill Climbing Method with Push Down Stack

The goal seeker is easily hampered by the deadlocked situation explained in Section 2.2 due to ill-structured constraints of the problem solving system. This section investigates a strategy equipped with a push down stack, which enables backtracking to avoid such a deadlocked situation. If the system meets the situation, it must return to a previous state (backtracking) to proceed in another direction. The stack saves information necessary for the backtracking. A strategy can be designed using a combination of the hill climbing method and a push down stack. This method is called PD method.

The target formulation of a user model is assumed to be given by

$$\textit{user model} = \langle A, C, Y, \delta, \lambda, \textit{genA}, \textit{constraint}, g, \textit{st}, c_0, C_f \rangle$$

where g is given as a function whose domain is the set of states.

This section investigates general properties of the goal seeker with PD method in Figure 6. Let me start by introducing some underlying concepts.

Definition 4.1 $legalAs:C\rightarrow \wp(A^*)$

Let us define a function, $legalAs:C\rightarrow \wp(A^*)$: for $c\in C$

$$\begin{aligned} a_1a_2...a_p\in legalAs(c) \text{ iff } & \delta(c, a_1)=c_1 \ \&\ a_1\in genA(c) \ \&\ constraint(c_1)=true \ \& \\ & \delta(c_1, a_2)=c_2 \ \&\ a_2\in genA(c_1) \ \&\ constraint(c_2)=true \ \& \\ & \vdots \\ & \delta(c_{p-1}, a_p)=c_p \ \&\ a_p\in genA(c_{p-1}) \ \&\ constraint(c_p)=true. \end{aligned}$$

Definition 4.2 Solvable state and unsolvable state

Let $c\in C$ be called a solvable state *iff*

$$(\exists\ \alpha\in legalAs(c))(\delta(c,\ \alpha)\in C_f).$$

Let $c\in C$ be called an unsolvable state *iff* c is not solvable, i.e.,

$$(\forall\ \alpha\in legalAs(c))(\delta(c,\ \alpha)\notin C_f).$$

Definition 4.3 Forward Layer Structure $\{C_p\}_p$ on C

A class of subsets,

$$\{C_p\}_p,\ C_p\subset C\ (p=0, 1, 2, ...)$$

will be defined recursively. Let

$$C_0=\{c_0\};$$

$$C_{p+1}=\{c'|(\exists\, c\in C_p)(\exists\, a\in A)(a\in genA(c)\ \&\ \delta(c, a)=c'\ \&\ constraint(c')=true)\}-\bigcup\{C_r|r\leq p\}.$$

The class $\{C_p\}_p$ is the forward layer structure on C.

Definition 4.4 Linear Order Relation $\leq$ on $\bigcup C_k$

Let $c, c'\in\bigcup C_k$ be arbitrary. Then, p and q are unique such that $c\in C_p$ and $c'\in C_q$. Let

$$c \le c' \leftrightarrow p \le q.$$

Definition 4.5 Restricted Allowable Activity Function of PD Method: $genA^{*}: \bigcup C_k \rightarrow \wp(A)$

Let a function,

$$genA^{*}: \bigcup C_k \rightarrow \wp(A)$$

be defined as follows: let $c \in C_k$. Because

$$i \neq j \rightarrow C_i \bigcap C_j = \phi,$$

there is a unique p such that $c \in C_p$. Then,

$$genA^{*}(c) = \{a \mid a \in genA(c) \ \& \ \delta(c, a) \in C_{p+1}\}.$$

genA* is the restricted allowable activity function of the PD method.

Definition 4.6 Solution Path and Efficient Solution Path

Let

$$\alpha \in legalAs(c_0)$$

be called a solution path *iff*

$$\delta(c_0, \alpha) \in C_f \subset C.$$

Let

$$\alpha = a_1 a_2 \ldots a_p \in legalAs(c_0)$$

be called an efficient solution path *iff* α is a solution path and satisfies the following relations:

$$\delta(c_0, a_1) = c_1 \in C_1,$$

$$\delta(c_1, a_2) = c_2 \in C_2,$$

$\vdots$

$\delta\,(c_{p-1}, a_p)=c_p \in C_p=C_f$

where for $i<p$, $C_i \neq C_f$.

The efficient solution path will be used as a desirability criterion for the goal-seeker of the PD method.

Definition 4.7 ***backtrack:*** $\boldsymbol{C \to \wp(A)}$

Let a function *backrack:*$C \to \wp(A)$ such that for any solvable $c \in C$

$a \in backtrack(c)$ *iff* $a \in genA(c)$ *and* $\delta\,(c, a)$ *is a solvable state.*

Although a goal is used to select an action to yield the next proper state on a C_k, it is not necessarily compatible with the order of C. However, if it is compatible, it will be shown to derive stronger results. We use the following compatibility condition.

Definition 4.8 Compatibility Condition for goal

A goal is said to satisfy a compatibility condition *iff* it satisfies the following relation: for c, $c' \in \bigcup C_p$

$goal(c) \geq goal(c') \leftrightarrow c \leq c'$.

Definition 4.9 Strategy of the Hill-climbing Method with Push Down Stack: $\boldsymbol{\sigma_{pd}: \bigcup C_p \to A}$

Let

$\sigma_{pd}: \bigcup C_p \to A$

be

$\sigma_{pd}(c)=\hat{a}$

where

$$\hat{a} \in backtrack(c)$$

and satisfies

$$min\{goal(\delta(c, a)) \mid a \in backtrack(c)\} = goal(\delta(c, \hat{a})).$$

Then, σ_{pd} will be called the strategy of the hill-climbing method with a push-down stack.

The goal seeker of this section is characterized by σ_{pd}. It should be noted that the definition of σ_{pd} is independent of the compatibility of the goal. σ_{pd} is not a simple feedback law because it requires a backtrack operation to assure δ(c, σ_{pd}(c)) is a solvable state.

Definition 4.10 Loop-free Strategy

A strategy σ:C$\rightarrow$A is called loop free if it satisfies

$$(\forall p, q)(p \neq q \rightarrow \pi_\sigma^p(c_0) \neq \pi_\sigma^q(c_0)).$$

where π_σ is the dynamical mapping of σ, i.e.,

$$\pi_\sigma(c) = \delta(c, \sigma(c))$$

and

$$\pi_\sigma^{p+1}(c_0) = \pi_\sigma(\pi_\sigma^p(c_0)).$$

The following is an obvious but fundamental fact for problem solving.

Theorem 4.1

Suppose C is finite and the initial state is solvable. Then, if a strategy σ is loop free, p exists such that $\pi_\sigma^p(c_0) \in C_f$ or it yields a solution.

Proof

Let

$$C^p = \{c_0, \pi_\sigma(c_0), \ldots, \pi_\sigma^p(c_0)\}.$$

If C^p=C,

$$C^p \cap C_f \neq \phi.$$

Suppose

$$C^p \neq C.$$

Because σ is loop free,

$$C^1 \subsetneq C^2 \subsetneq \ldots \subsetneq C^p.$$

Because C is finite,

$$C^p \cap C_f \neq \phi$$

for some p.

Q.E.D.

Proposition 4.1

Let $c \in C_p$ be arbitrary. Then

$$(\forall \alpha)(\exists q)(\alpha \in legalAs(c) \ \& \ \delta(c, \alpha)=c' \rightarrow c' \in Cq).$$

Proof

We use the mathematical induction.

(i) Suppose

$$\alpha = a \in legalAs(c).$$

Then

$$a \in genA(c)$$

and

$$constraint(\delta(c, a))=true.$$

Let

$$c' = \delta(c, a).$$

Due to the definition of C_{p+1}

$$c' \in C_{p+1} \vee c' \in \bigcup \{C_r \mid r \leq p\}.$$

If

$$c' \notin C_{p+1},$$

then

$$(\exists r)(c' \in C_r).$$

(ii) Suppose if

$$length(\alpha) \leq l,$$

$$\delta(c, \alpha) \in C_q$$

for some *q*. Let

$$\beta = \alpha a$$

and

$$length(\beta)=l+1$$

where

$$a \in genA(\delta(c, \alpha))$$

is arbitrary. The induction hypothesis implies

$$\delta(c, \alpha) \in C_q$$

for some q. Then, the argument (i) implies that

$$\delta(\delta(c, \alpha), a) \in C_{q+1}$$

or

$$\delta(\delta(c, \alpha), a) \in C_r.$$

Q.E.D.

The following is also a fundamental fact for the PD method.

Proposition 4.2

Let $\alpha \in legalAs(c_0)$ be arbitrary. Let $c' = \delta(c_0, \alpha)$. Then

$$(\exists \alpha' \in legalAs(c_0))(\alpha' = a_1 \ldots a_k \ \& \ \delta(c_0, a_1 \ldots a_k) \in C_i \text{ for each } i \ \& \ \delta(c_0, \alpha') = c').$$

Proof

Because of Proposition 4.1, k exists such that

$$\delta(c_0, \alpha) = c' \in C_k.$$

Because

$$c' \in C_k = \{c' \mid a \in genA(c) \ \& \ \delta(c, a) = c' \ \& \ c \in C_{k-1} \ \& \ constraint(c') = true\} - \bigcup\{C_r \mid r \le k-1\},$$

$$c' = \delta(c_{k-1}, a_k) \ \& \ c_{k-1} \in C_{k-1} \ \& \ a_k \in genA(c_{k-1}) \ \& \ constraint(c') = true.$$

Because

$$c_{k-1} \in C_{k-1},$$

c_{k-2} and a_{k-1} exist such that

$$c_{k-1}=\delta(c_{k-2}, a_{k-1}) \;\&\; c_{k-2}\in C_{k-2} \;\&\; a_{k-1}\in genA(c_{k-2}) \;\&\; constraint(c_{k-1})=true.$$

In this way we can find $a_1, a_2, ..., a_k, c_1, ..., c_{k-1}$ such that

$$\delta(c_0, a_1)=c_1\in C_1 \;\&\; a_1\in genA(c_0) \;\&\; constraint(c_1)=true,$$

$$\delta(c_1, a_2)=c_2\in C_2 \;\&\; a_2\in genA(c_1) \;\&\; constraint(c_2)=true,$$

$$\vdots$$

$$\delta(c_{k-1}, a_k)=c'\in C_k \;\&\; a_k\in genA(c_{k-1}) \;\&\; constraint(c')=true.$$

Consequently,

$$\alpha'=a_1...a_k\in legalAs(c_0),\; \delta(c_0, a_1...a_i)\in C_i$$

and

$$\delta(c_0, \alpha')=c'.$$

Q.E.D.

Theorem 4.2

There is a solution path iff an efficient solution path exists.

Proof

The if part is obvious. The only if part comes from Proposition 4.2 where

$$c'\in C_f.$$

Q.E.D.

We will characterize an efficient solution path. The following should be obvious.

Corollary 4.1

Suppose

$\alpha = a_1 a_2 ... a_p \in legalAs(c_0)$

is a solution path. Suppose α satisfies the following relations:

$\delta(c_0, a_1) = c_1 \notin C_0,$

$\delta(c_1, a_2) = c_2 \notin C_0 \cup C_1,$

$\vdots$

$\delta(c_{p-1}, a_p) = c_p \notin C_p \cup \{C_k \mid k<p\}.$

Then, α is an efficient solution path.

An efficient solution path is a solution path that never goes backward (with respect to the order of C) on the way to a target state. The following should be also obvious.

Corollary 4.2

Suppose

$\alpha = a_1 a_2 ... a_p \in legalAs(c_0)$

is a solution path. Suppose α satisfies the following relations:

$a_1 \in genA^*(c_0),$

$a_2 \in genA^*(c_1)$ *where* $\delta(c_0, a_1) = c_1,$

$a_3 \in genA^*(c_2)$ *where* $\delta(c_1, a_2) = c_2,$

$\vdots$

$a_p \in genA^*(c_{p-1})$ *where* $\delta(c_{p-2}, a_{p-1}) = c_{p-1}.$

Then, α is an efficient solution path.

Q.E.D.

When the goal is compatible, we have strong results for σ_{pd}.

Proposition 4.3

Let $c \in C_p$ be solvable. Suppose the goal is compatible. Let $\hat{a} \in \sigma_{\mathrm{pd}}(c)$ where σ_{pd} is a strategy of the PD method. Then,

$$\delta(c, \hat{a}) \in C_{p+1}.$$

Proof

Because c is a solvable state, Proposition 4.2 asserts that there is

$$\alpha = a_1 \ldots a_k \in legalAs(c)$$

such that

$$\delta(c, a_1) = c_1 \in C_{p+1},$$

$$\delta(c_1, a_2) = c_2 \in C_{p+2},$$

$$\vdots$$

$$\delta(c_{k-1}, a_k) = c_k \in C_f.$$

Because

$$a_1 \in genA(c)$$

and δ (c, a_1) is a solvable state,

$$c < \delta(c, a_1)$$

implies

$$goal(c) \geq goal(\delta(c, a_1)) \geq goal(\delta(c, \hat{a})),$$

that is,

$c < \delta(c, \hat{a})$.

Then, the definition C_{p+1} implies that

$\delta(c, \hat{a}) \in C_{p+1}$.

Q.E.D.

Corollary 4.3

If the goal is compatible, σ_{pd} is loop free.

Corollary 4.3 indicates that if the goal is compatible and if the strategy is σ_{pd}, the usual technique of problem solving that the trajectory of states is saved to avoid repetition of states is not necessary.

Theorem 4.3

σ_{pd} is the PD method always yields an efficient solution path if the initial state is a solvable state and the goal is compatible.

In practice there are two ways to derive an efficient solution path. The first way is to use Corollary 4.1, which says that an efficient solution path is produced if a next state c_i is found outside of $\bigcup\{C_p|p<i\}$ for every i. However, because it is usual to consume a large amount of memory to memorize $\bigcup\{C_p|p<i\}$, Corollary 4.1 has difficulty with respect to computation speed and memory limitation. Then, if we use a heuristic that the history H of the visited states may cover an essential part of $\bigcup\{C_p|p<i\}$, H may be substituted for $\bigcup\{C_p|p<i\}$. In general, this heuristic is useful because in any case H must be saved in order to make a strategy loop free.

The second way to derive an efficient solution path is to use Theorem 4.3. In that case, the goal is required to be compatible. Although the compatibility condition might be considered very exceptional, it is not necessarily true. If a target problem belongs to the class of implicit solving actions and $a:Y \to Y$ is selected as an action to improve an output y, and if the goal is defined to evaluate the improvement, the compatibility condition holds. The data-mining system problem, for example, corresponds to this case.

4.2 Total Process of Standardized Goal Seeker

Figure 11 shows the total process of a general goal-seeker based on Corollary 4.1 and Theorem 4.3.

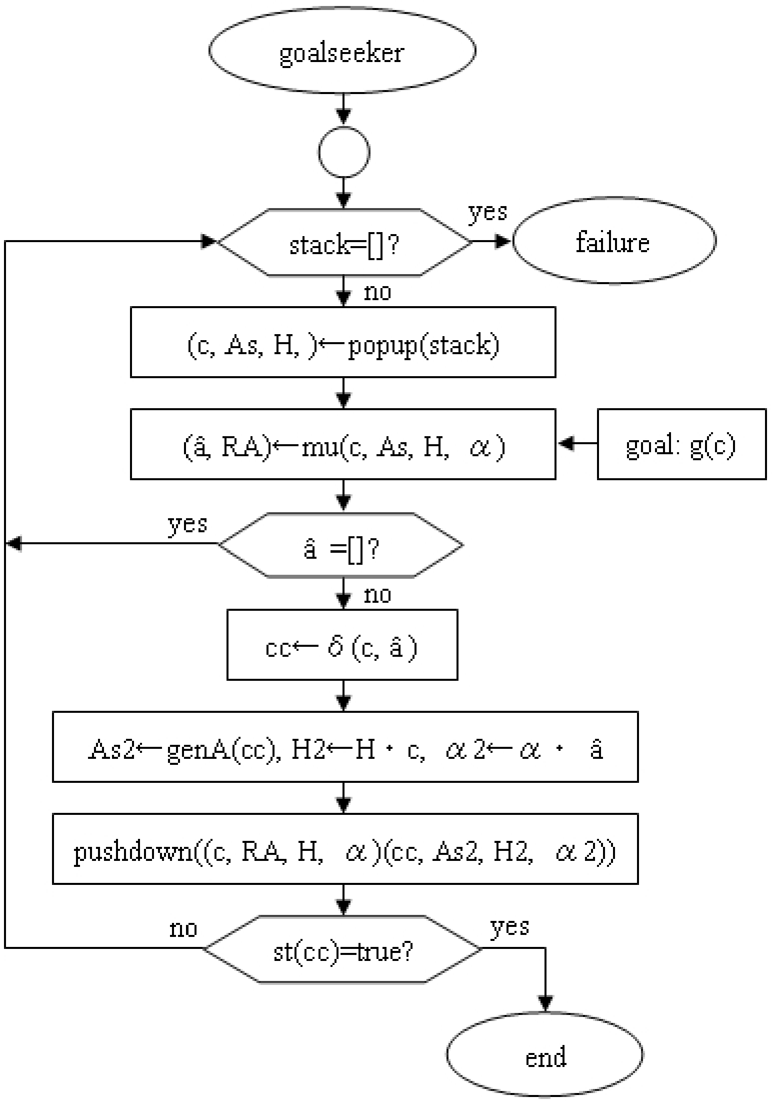


Figure 11. Total process of goal-seeker based on Corollary 5.1 and Theorem 5.3.

All necessary information to yield an action $\sigma_{pd}(c)$ is stored in a push down stack. The goal-seeker, first, tries to pop up the top information. If the trial fails, in general it indicates failure of the goal-seeker. However, in the case of an open target, it may indicate that the goal-seeker has reached a final state (Refer to the definition of C_f in Chapter 3). In either case, the operation of the goal-seeker must finish. If the trial is successful, the goal-seeker gets (c, As, H, α), current state, set of available actions $\subset$ genA(c), the history of the visited states, and the history of selected actions as the top information of the stack. It then selects ($\hat{a}$, RA) where $\hat{a}$ is assumed to satisfy the relation

$$min\{goal(\delta(c, a)) \mid a \in As\} = goal(\delta(c, \hat{a}))$$

and

$$RA = As - \{\hat{a}\}.$$

δ(c, $\hat{a}$) must not be equal to an element of H (refer to Theorem 4.3). If c is a deadlocked state, $\hat{a}$=[] is assumed. In this case the goal-seeker must do the backtrack operation, that is, it pops up the next information from the stack. This backtrack operation ultimately assures the condition "δ(c, $\hat{a}$) is a solvable state" or $\hat{a} \in$ backtrack(c) holds and hence $\hat{a} = \sigma_{pd}$(c).

If $\hat{a}$ is a regular action ($\hat{a} \neq$ []), the goal-seeker applies $\hat{a}$ to the process δ to get the next state cc=δ(c, $\hat{a}$). It generates the set of available activities, As2 for the next state, cc. The goal-seeker saves (push down) two pieces of information, (c, RA, H, α) and (cc, As2, H2, α2) where H2=$H \cdot c$ and α2=$\alpha \cdot \hat{a}$, i.e., the current state c and the action $\hat{a}$ are appended to H and α, respectively. The former information, (c, RA, H, α), will be used when the goal-seeker backtracks to the state c. The latter information is used at the next state cc. Then, the stopping condition, st(), is checked. If st(cc)=true, the total process must finish. If st(cc)$\neq$true, it goes to the pop up stage.

The total extSLV, which consists of the goal-seeker and the process, δ, is formalized as a generator:

$$extSLV = \langle Stack, solProc, st, (c_0, genA(c_0), [], []) \rangle$$

where *Stack* is the state set of *extSLV* and $(c_0, genA(c_0), [], [])$ is an initial state. *Stack*, $solProck{:}Stack \rightarrow Stack$ and $st{:}C \rightarrow \{true, false\}$ are defined as below.

Let

$\Gamma = C \times \wp(A) \times H \times Sol$; *stack element,*

$H = C^*$; *history of states,*

$Sol = A^*$; *history of actions,*

$Stack = \Gamma^*$.

Let $mu{:}\ \Gamma \rightarrow A \times \wp(A)$ be

$mu(c, As, \varphi, \alpha) =$

$$\begin{cases} (\hat{a}, As - \{\hat{a}\})\ if\ min\{goal(\delta(c,a)) \mid a \in As\ \&\ constraint(\delta(c,a)) = true\} \\ \qquad\qquad\qquad = goal(\delta(c,\hat{a})) \\ ([], []) \qquad o.w. \end{cases}$$

Then, $\sigma{:}C \rightarrow A$ is defined as the function that for each state, *c*, gives the action, $\hat{a}$, which is given by the function, *mu*. In the definition of mu, the condition $\hat{a} \in backtrack(c)$ is not required because the condition "$\delta(c, \hat{a})$ is a solvable state" is supported by the backtracking as mentioned above. Then, $solProck{:}Stack \rightarrow Stack$ *is*

$$solProc(\gamma^* \gamma_n) = \begin{cases} \gamma^* \gamma_n \gamma_{n+1} & if\ mu(\gamma_n) = (\hat{a}, RA)\ and\ \hat{a} \neq [] \\ \gamma^* & o.w. \end{cases}$$

where

$\gamma_n = (c, As, \varphi, \alpha)$,

$\gamma_n{}' = (c, As - \{\hat{a}\}, \varphi, \alpha)$,

$\gamma_{n+1} = (\delta(c, \hat{a})), \varphi \cdot c, \alpha \cdot \hat{a})$.

$st{:}C \rightarrow \{true, false\}$ is

$st(c) = true \leftrightarrow c \in C_f$.

The generator is actually a push down automaton without input.

4.3 Implementation in extProlog

Implementation of extSLV is made by describing the user model,

$<A, C, \delta, \lambda, genA, constraint, goal, c_0, C_f>$,

which is dependent from problems, in extProlog, and attaching extSLV generator formulation,

$<Stack, solProc, st, (c_0, genA(c_0), [], [])>$,

which is coded in extProlog and is independent from problems [16](refer to Appendix I).

The total program of the extSLV whose strategy is the PD method is listed as following:

```
# include "stdPDsolver.p";

delta():-...;

lambda()-...;

genA():-...;

constraint():-...;

initialstate() ;

finalstate();

goal():-...;

st():-...;
```

From *delta():-...;* to *st():-...;* are rules which constitute the user model, that is, the implementation of the dynamic optimization problem. *#include* “stdPDsolver.p”; is the statement which calls the goal seeker defined in Chapter 4. It is independent from problems.

The stdPDsolver.p consists of the following rules:

```
stdPDsolver():-…;

solProc():-…;

goalseeker():-…;

mu():-…;

sigma():-…;
```

stdPDsolver() is the entrance of stdPDsolver.p. It sets some default values and calls *solProc()*. The *mu()* is the implementation of μ. The *sigma()* is the implementation of σ in Figure 11, and is the program which actually do the hill climbing in the *mu()*.

Those rules are represented as sets, functions and relation, and they are respectively implemented as lists, predicates and special predicates. Although sets do not have orders among their elements, lists have ones. For that reason, sorting them in a standard order is required in order to check equivalence among sets.

Appendix I shows the extSLV of WGC problem, which is the extProlog description directly converted from the set theoretic description. For example, the term "If $\{f, a\}\subset C$, then $\delta(c, a)=c-\{f, a\}$ where a≠f." is described as following:

```
delta(C,A,C2):-
        A<>f,
        subset([f, A], C),
        sort(minus(C,[f, A]),C2);
```

stdPDsolver.p can also be given by translating its set theoretic description in Chapter 4 into the extProlog code. Practical efficiency to enhance a computing speed is made (this is not a tuning for each problem). The Appendix includes the whole codes of stdPDsolver.p.

Chapter 5 Applications: WGC Problem and Traveling Salesman Problem

This chapter shows two applications of The Model Theory Approach: The WGC Problem and The Traveling Salesman Problem.

The WGC Problem and Traveling Salesman Problem, which will be described in Section 5.1 and 5.2 respectively, are famous and typical examples of problem solving.

5.1 WGC Problem

A farmer, a goat, a wolf, and a cabbage have to cross a river. A boat nearby only has enough room for the farmer and one other thing. In addition, the wolf eats the goat and the goat eats the cabbage unless the farmer is with them. In this constrained condition, what is the fewest number of trips he must take so that the goat does not eat the cabbage, and so the wolf does not eat the goat?

This is called the WGC Problem. It cannot be said that it is a real management problem, but it can be said that it has the same structure as the management problem has because it is a scheduling problem under the given constraints (rules).

The WGC Problem is categorized as E-O-C problem, that is, its solution is expressed as a sequence of actions of the solution process, the evaluation of the output is not given by the problem specification environment, and a final state of the solving process can be determined when the process, δ, is specified.

a. Stage 1: Input-output Block Diagram

Figure 12 illustrates an input-output block diagram of the WGC Problem.

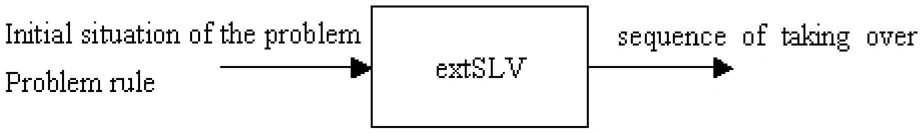


Figure 12. Input-output block diagram for the WGC Problem

The input is represented as the initial situation and rules of WGC Problem as described above. On the other hand, the output is the order of the taking a wolf, a goat and a cabbage over the river.

b. Stage 2: Input-output Specification in Set Theory

First of all, define set A as

$$A=\{f, w, g, ca\}$$

where f, w, g and ca respectively refer to farmer, wolf, goat and cabbage. For these elements, the elements of the following set, *cons*, represent the constrained conditions on either bank of a river:

$$cons=\{\{w, g, ca\},\{w, g\},\{g, ca\}\}\subset\wp(\mathrm{A}) .$$

$\wp(\mathrm{A})$ is a power set of A.

The rule of the game is represented as an automaton as follows. Suppose the game is started from the left bank.

Suppose the state set, C_g, of the game as

$$C_g=\wp(\mathrm{A}) .$$

A state $c\in C_g$ indicates the collection of the elements of A, which are on the left bank, and this is the state of the game. As a consequence, the initial state can be given as

$c_0=A.$

Then, for the state set, C_g, the state transition function

$\delta_g: C_g \times A \rightarrow C_g$

is defined as follows:

$$\begin{aligned}
&\delta_g (c, a)=c_2 \leftrightarrow \\
&\quad (a \neq f \text{ and } f \in c \text{ and } a \in c) \rightarrow (c_{20}=c-\{f, a\}), \\
&\quad (a \neq f \text{ and } f \notin c \text{ and } a \notin c) \rightarrow (c_{20}=c \cup \{f, a\}), \\
&\quad (a=f \text{ and } f \in c) \rightarrow (c_{20}=c-\{f\}), \\
&\quad (a=f \text{ and } f \notin c) \rightarrow (c_{20}=c \cup \{f\}), \\
&\quad constraint(c_{20})=true;
\end{aligned}$$

The first two lines are the case where the farmer takes $a \in c$ on the boat. In contrast, the last two lines are the case where the farmer crosses alone. Also, the first and third lines are the case where the former is on the left bank, and the second and third lines are the case where he is on the right side.

For example, when the farmer and *a* are on the left bank, (i.e. $\{f, a\} \subset c$),

$\delta_g(c, a)=c-\{f, a\}$

indicates that the farmer crosses the river with a.

The output function

$\lambda_g: C_g \times A \rightarrow A$

is defined as

$\lambda_g(c, a)=a.$

Then, the problem specification environment of the WGC Problem is given as the following structure:

$$<A, \wp(A), cons, \delta_g, \lambda_g, c_0>$$

and the output set Y is given as

$$Y=A^*$$

where A^* is the free monoid of A. For instance, the elements $g \cdot f \cdot w \in A^*$ represents "the farmer firstly crossed the river with the goat, he went back to the opposite bank alone, and then finally he took the wolf over the river".

c. Stage 3: Process Specification as Automaton

Because the WGC problem belongs the E-O-C class, suppose the action set as the set, A, the output set, Y, as A^*, and the state set as Y, then, the state transition function

$$\delta^*: Y \times A \to Y$$

can be defined as

$$\delta^*(y, a) = y \cdot a$$

where $y \cdot a$ is the concatenation of y and a. δ^* is attached * because of being distinguished from δ mentioned later.

The output function

$$\lambda^* : Y \times A \to A$$

is defined as

$$\lambda^*(y, a)=a$$

In the WGC Problem, consider the state set of the game, C_g, as the desirable state set of the problem solving process, then suppose a reduction mapping

$$\omega: Y \to C_g$$

as

$$\omega(y) = \delta_g(A, y) = \text{“elements that exist on the left bank when } y \in A^* \text{ is performed”}$$

where δ_g is the natural extension of the above-mentioned

$$\delta_g: C_g \times A \to C_g$$

into

$$\delta_g: C_g \times A^* \to C_g.$$

In this case, for all y and a,

$$\omega(y) = \omega(y') \to \omega(y \cdot a) = \omega(y' \cdot a)$$

is approved. In other words, in the case of WGC Problem, we can use

$$<\delta_g, \lambda_g>$$

as the automaton representation of the problem solving concept. Hereinafter, I use δ, λ and C for the state transition function, the output function and the state set, respectively, for the sake of the notational convenience.

Two more functions,

$$genA: C \to \wp(A)$$

and

$$constraint: C \to \{true, false\},$$

are needed to be given in this stage. The former of the wgc problem is defined as

$genA(c)=As \leftrightarrow$

$(f \in c) \rightarrow (As = c)$ *otherwise* $(As = A-c);$

The latter is given as following from the relation cons:

```
$constraint(c)=true \leftrightarrow$
    $(f \in c) \rightarrow ($
        $A-c \notin cons$
    $)$ otherwise $($
        $c \notin cons$
    $);$
```

$cons=\{\{w, g, ca\},\{w, g\},\{g, ca\}\};$

The initial state, c_0 can be

$c_0= \delta_g(A, \Lambda)=A.$

Next, we need to give the representation of the target state. The wgc problem is the closed target, and it is the state that all things were taken over to the right bank, and can be represented as

$C_f=\{\phi\}.$

d. Stage 4: Dynamic Optimization Formulation

At this stage, the evaluation function,

$goal: C \rightarrow Re,$

is required to be given. Because the WGC Problem belongs to the E-O-C class, we must give it heuristically. We defined it as

$$goal(c)=|c|$$

where |c| indicates the cardinality of c.

As above, the user model was formulated as the dynamic optimization problem as following:

$$user\ model\ = <A, C, \delta, \lambda, genA, constraint, goal, c_0, C_f>.$$

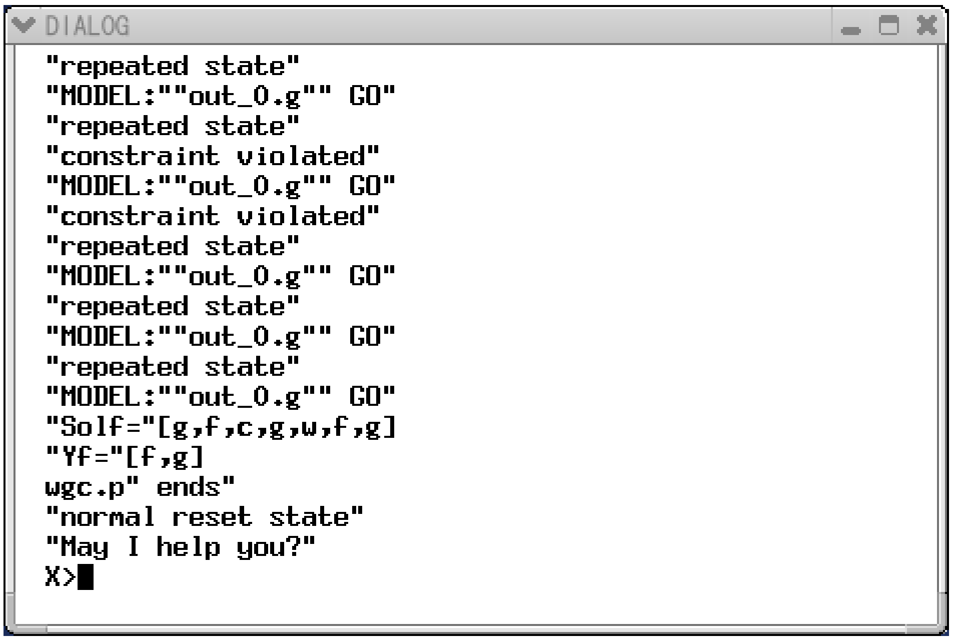
e. Stage 5: Implementation in extProlog

The final stage is the implementation in the extProlog. At this stage, we transform the set theoretically formulated user model into the extProlog code.

In Appendix II, the whole code of the extSLV for WGC Problem is listed.

f. Result of Implementation

Figure 13 shows a solution (output of the extSLV) produced by the extSLV for the WGC Problem.



```
"repeated state"
"MODEL:""out_0.g"" GO"
"repeated state"
"constraint violated"
"MODEL:""out_0.g"" GO"
"constraint violated"
"repeated state"
"MODEL:""out_0.g"" GO"
"repeated state"
"MODEL:""out_0.g"" GO"
"repeated state"
"MODEL:""out_0.g"" GO"
"Solf="[g,f,c,g,w,f,g]
"Yf="[f,g]
wgc.p" ends"
"normal reset state"
"May I help you?"
X>█
```

Figure 13. Solution by WGC Problem extSLV

"[g, f, c, g, w, f, g]" was the solution that the extSLV found.

Figure 14 displays the dynamic behavior of the goal.

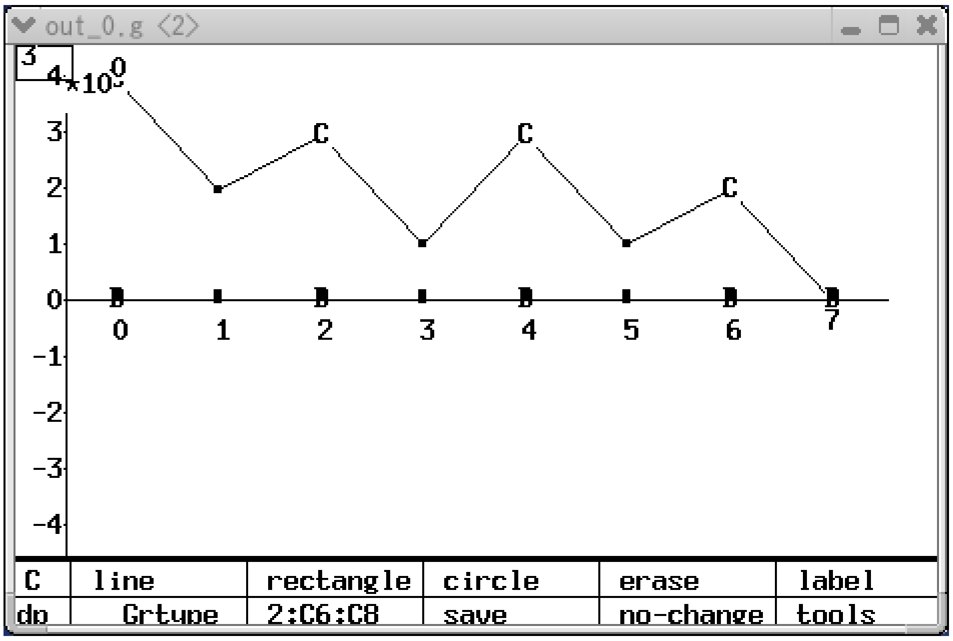


Figure 14. Dynamic procedure of goal

You can see the way that the value of the goal is decreasing and finally got to 0. That was because the goal was defined as the number of things on the left side of the river.

5.2 Traveling Salesman Problem

Given a collection of cities and the distances of travel between each pair of them, the traveling salesman problem, or TSP for short, is to find the shortest way of visiting all of the cities and returning to your starting point. In the standard version we study, the travel distances are symmetric in the sense that traveling from city X to city Y is the same distance as traveling from Y to X.

The Traveling Salesman Problem is categorized as an E-C-C problem, that is, its solution is expressed as a sequence of actions of the solution process, the evaluation of the output is given by the problem specification environment, and a final state of the solving process can be determined when the process, δ, is specified.

a. Stage 1: Input-output Block Diagram

Figure 15 illustrates an input-output block diagram of the Traveling Salesman Problem.

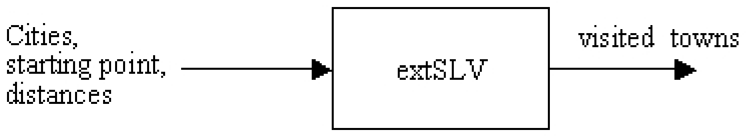


Figure 15. Input-output block diagram for Traveling Salesman Problem

The input is represented as a set of cities traveled, distances between those connections, and a city of a starting point. On the other hand, the output is sequences of cities traveled to.

b. Stage 2: Input-output Specification in Set Theory

First of all, define a set of cities, *Cities*, as

$$Cities=\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$$

In the example dealt with in this thesis, suppose that the salesman travels 15 cities. The cities are numbered from 1 to 15. Besides, the distances between all of two cities in Cities are defined as a set,

$$DisList=Re\times Re^{2}\times\cdots\times Re^{15},$$

such as

$$\begin{aligned}DisList=\{&\{0\},\\ &\{123, 0\},\\ &\{56, 25, 0\},\\ &\{78, 98, 225, 0\},\\ &\{66, 908, 140, 25, 0\},\\ &\{37, 345, 87, 678, 336, 0\},\\ &\{999, 41, 54, 402, 34, 134, 0\},\\ &\{79, 78, 89, 84, 755, 62, 244, 0\},\end{aligned}$$

{30, 94, 65, 44, 98, 79, 745, 126, 0},

{74,55,65,110,100,80,96,864,200,0},

{61,317,58,531,180,39,356,67,46,67,0},

{88,173,20,970,33,40,678,116,806,19,79,0},

{138,57,99,340,15,78,10,397,64,76,80,120,0},

{68,468,15,60,70,198,34,305,79,118,19,369,777,0},

{105,56,881,576,29,378,169,58,41,60,87,119,12,22,0}};

For example, the element, 0, which is only one element of the first element of *disList* indicates the distance from city 1 to city 1, that's why it is zero. In this set, the distance between city 1 and city 2, for instance, is 123 kilometers. In addition, the starting point is city 1, that is,

$city_0=1.$

Then, the problem specification environment of the Traveling Salesman Problem is given as the following structure:

$<Cities, DisList, city_0>,$

and the output set Y is given as

$Y=Cities^*$

where $Cities^*$ is the free monoid of *Cities*.

c. Stage 3: Process Specification as Automaton

Because the Traveling Salesman Problem belongs to the E-C-C class, suppose the action set as the set,

$A=Cities,$

the output set can be defined as

$Y=A^{*}$.

In the Traveling Salesman Problem, Y can be used as the state set, that is,

$C=Y$.

Then, the state transition function,

$\delta : C \times A \rightarrow C,$

can be defined as

$\delta\ (c, a) = c \cdot a$

where $c \cdot a$ is the concatenation of c and a.

The output function

$\lambda : C \times A \rightarrow A$

is defined as

$\lambda(c, a)=a$

Two more functions,

$genA:C \rightarrow \wp(A)$

and

$constraint:C \rightarrow \{true, false\},$

are needed to be given in this stage. The former of this problem is defined as

$genA(c)=A-c$

because the applicable action candidates for the current city can be the cities which the salesman has not visited yes.

The constraint on the states is only that the salesman cannot visit the same city twice. It is impossible that he visits the same city twice under the above definition of *genA*. Therefore, the function, constraint, is given as

$constraint(c)=true.$

The initial state, c_0 can be

$c_0=city_0=1.$

Next, we need to give the representation of the target state. The Traveling Salesman Problem is the closed target, and it is the state that the salesman visits all cities, and can be represented as

$C_f=\{c \mid |c|=|A|\}.$

d. Stage 4: Dynamic Optimization Formulation

At this stage, the evaluation function,

$goal: C \rightarrow Re,$

is required to be given. Traveling Salesman Problem is an E-C-C problem, so it has a closed goal, and it is defined as the total distance of a route, c. However, because the function, goal, is used to determine the next city to visit, all it needs to do is to assign the distance between the last two cities in c to a given state, c. It is given as follows:

```
$goal(c)=r \leftrightarrow (|c|=1) \rightarrow (r=0)$ otherwise (
    $city_1=project(c, 0),$
    $vcities = c - \{city_1\},$
    $city_2 = project(vcities, 0),$
    $r = distance(city_1, city_2));$
```

where project(S, n) is the function that assign the nth element of a set, S, and distance($city_1$, $city_2$) is the function that assign the distance between $city_1$ and $city_2$. Speaking of the function, project, except in the case where n is 0, it assigns the last element of the given set. Those two functions are defined as follows:

```
$project(\{e_1, e_2, ..., e_n\}, i)=e \leftrightarrow$
    $(i=0) \rightarrow ($
        $e=e_n$
    $)$ otherwise $($
        $e=e_i$
    $);$

$distance(city_1, city_2)=r \leftrightarrow$
    $(city_1<=city_2) \rightarrow ($
        $DisCity=project(DisList, city_2),$
        $R=project(DisCity, city_1)$
    $)$ otherwise $($
        $DisCity=project(DisList, city_1),$
        $R=project(DisCity, city_2)$
    $);$
```

As above, the user model was formulated as the dynamic optimization problem as follows:

$$user\ model = <A, C, \delta, \lambda, genA, constraint, goal, c_0, C_f>.$$

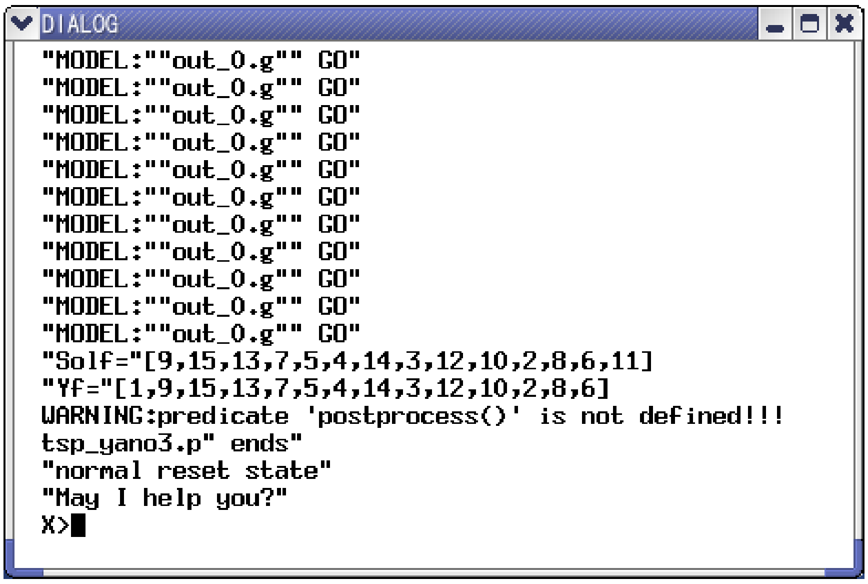
e. Stage 5: Implementation in extProlog

The final stage is the implementation in the extProlog. At this stage, we transform the set theoretically formulated user model into the extProlog code.

In Appendix III, the whole code of the extSLV for the Traveling Salesman Problem is listed.

f. Result of Implementation

Figure 16 shows a solution (output of the extSLV) produced by the extSLV for the Traveling Salesman Problem.



```
"MODEL:""out_0.g"" GO"
"MODEL:""out_0.g"" GO"
"MODEL:""out_0.g"" GO"
"MODEL:""out_0.g"" GO"
"MODEL:""out_0.g"" GO"
"MODEL:""out_0.g"" GO"
"MODEL:""out_0.g"" GO"
"MODEL:""out_0.g"" GO"
"MODEL:""out_0.g"" GO"
"MODEL:""out_0.g"" GO"
"MODEL:""out_0.g"" GO"
"Solf="[9,15,13,7,5,4,14,3,12,10,2,8,6,11]
"Yf="[1,9,15,13,7,5,4,14,3,12,10,2,8,6]
WARNING:predicate 'postprocess()' is not defined!!!
tsp_yano3.p" ends"
"normal reset state"
"May I help you?"
X>
```

Figure 16. Solution by the extSLV for Traveling Salesman Problem

"[9, 15, 13, 7, 5, 4, 14, 3, 12, 10, 2, 8, 6, 11]" was the solution that the extSLV found.

Figure 17 displays the dynamic behavior of the goal.

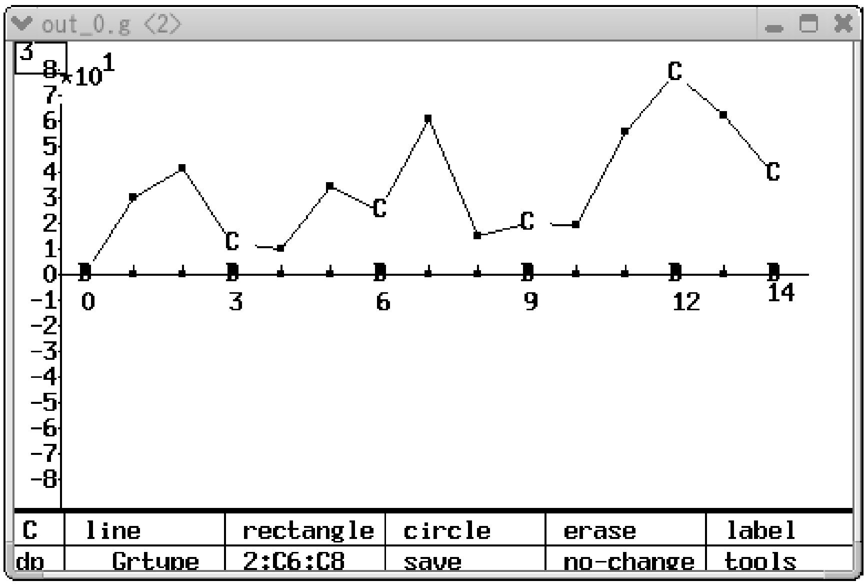


Figure 17. Dynamic procedure of goal

Each point on the graph represents the distance between the current city and the last.

Chapter 6 Conclusions

This dissertation discussed the design and implementation methodology of the extSLV. In particular, a systems approach was applied to the method. It revealed the structure of the extSLV (a problem solving system), and also showed its formulation and implementation.

The research achievements were the design and implementation procedure of the extSLV illustrated by Figure 8 and the formulation of the extSLV in Chapter 3 and 4.

This research clearly showed at least three advantages of the formal systems approach.

Firstly, we could propose the classification of the target problems as well as the development procedure of the extSLV. Although the current problem classification is at a preliminary stage, it can still give us a clue how to handle a new problem.

Secondly, as a result of revealing the structure of the extSLV, it could be decomposed into two parts, the problem dependent part and the problem independent part. The former is the process part of the extSLV, which is called the user model, and the latter is the goal seeker. We could provide the standardized one for the goal seeker. It could make a big contribution to rapid systems development. Since the design of the goal seeker is the most difficult part, it is quite helpful that we do not have to design it but can use a standardized one for it.

Thirdly, due to the narrow semantic gap between a set theoretic description and extProlog implementation, the Set Theory oriented formal approach results in rapid systems development.

Although the effectiveness of the approach depends on how applicable to the meaningful problems, this dissertation used the simple problems for example in order to achieve the understanding of the design and implementation procedure of the extSLV in the Model Theory Approach.

In order to make the approach more useful, we should enhance the knowledge about the extSLV and refine the classification of the problems in Section 2.4 by applying the approach to many meaningful problems.

Appendix I extProlog Code of Standardized Goal Seeker

```
/*stdPDsolver.p*/
stdPDsolver():-
  if preprocess() then
  else
    Dummy:=1
  end,
  stdPDsolver0(),
  if postprocess() then
  else
    Dummy2:=1
  end;
stdPDsolver0():-
  delGamma(0),
  assign(gammaId,0),
  initialstate(C0),
  if genA(C0,As0) then
  else
    xwriteln(0,"I guess your genA has some problem"),
    fail
  end,
  goal(C0,V0),
  assign(_regV,[V0]),
  getGammaId(GId),
  saveGamma(GId,[C0,As0,[],[]]),
  solProc(C0,[GId]),
  _regStack(Stack2),
  retract([_regStack,_regC]),
  project(Stack2,1,GId2),
  loadGamma(GId2,[Yf,Asf,Solf,Hf]),
  assign(_Solf,Solf),
  xwriteln(0,"Solf=",Solf),
  assign(_Yf,Yf),
  xwriteln(0,"Yf=",Yf),!;
saveGamma(GId,D):-
  switch(gamma(floor(GId/10),GId,D),GI),
  assert(GI);
loadGamma(GId,D):-
  switch(gamma(floor(GId/10),GId,D),GI),
  GI,!;
delGamma(I):-
  concat([gamma,".",I],GIT),
  parse(GIT,[GI0]),
  univ(GI,[GI0,Id,D]),
  if GI then
    retract([GI0]),
    delGamma(I+1)
  end;
```

```
solProc(C0,Stack0):-
  assign(_regStack,Stack0),
  assign(_regC,C0),
  repeat,
    _regStack(Stack1),
    _regC(C1),
    /*CC1:there is a backtrack*/
    goalseeker(C1,Stack1,CC1,A,C2,Stack2),
    if C2<>cfail then
      goal(C2,V2),
      _regV(L),
      append(L,[V2],L2),
      Len:=strlen(L2),
      if Len>200 then
        project(L2,["r",200,Len],L22)
      else
        L22:=L2
      end,
      assign(_regV,L22),
      show1(L22,plot)
    end,
    if C2<>cfail and st(C2) then
      Done:=1
    else
      Done:=0
    end,
    assign(_regStack,Stack2),
    if Stack2=[] then
```

```
      xwriteln(0,"NO NEXT JOB!!!")
    end,


    assign(_regC,C2),
    Done=1 or Stack2=[],!;


goalseeker(C,[G|Gs],CC,A,C2,Stack2):-
if status(trace,-3) then //old version
  xwriteln(0,"****************************"),
  xwriteln(0,"[state C   ]=",C),stop
end,
  mu(C,G,CC,A,C2,Gs2),
  append(Gs2,Gs,Stack2);


mu(cfail,G,CC,A,C2,Gs):-!,
  loadGamma(G,[CC,AAs,SSol,HH]),
  if status(trace,-3) then //old version
    xwriteln(0,"[state C   ]=",CC),stop
  end,
  if HH=[] then
    H:=[]
  else
    project(HH,-1,H)
  end,
  if SSol=[] then
    Sol:=[]
  else
    project(SSol,"head",Sol)
  end,
  if status(trace,-3) then   //old version
    xwriteln(0,"[AAs       ]=",AAs),stop
  end,
  sigma([CC,AAs,Sol,H],A,C2,RA),
  if status(trace,-3) then //old version
    xwriteln(0,"[selected A]=",A),
    xwriteln(0,"[next C    ]=",C2),
    xwriteln(0,"[RA        ]=",RA),stop
  end,
  if A<>[] then
    /*this is necessary for datamining;state is saved as a global variable*/
    delta(C,A,CC2),
    append(Sol,[A],Sol2),
    append([CC],H,H2),
    getGammaId(GId),
    saveGamma(GId,[CC,RA,Sol2,H2]),
    Gs:=[GId]
  else
    Gs:=[]
  end;

mu(C,G,C,A,C2,Gs):-
  loadGamma(G,[CC,AAs,SSol,HH]),
  if genA(C,As) then
  else
    xwriteln(0,"I guess yor genA has a problem"),
    fail
  end,
  if status(trace,-3) then //old version
    xwriteln(0,"[As        ]=",As),stop
  end,
  sigma([C,As,SSol,HH],A,C2,RA),
  if status(trace,-3) then //old version
    xwriteln(0,"[selected A]=",A),
    xwriteln(0,"[next C    ]=",C2),
    xwriteln(0,"[RA        ]=",RA),stop
  end,
  if A<>[] then
    /*this is necessary for datamining;state is saved as a global variable*/
    delta(C,A,CC2),
    append(SSol,[A],Sol2),
    append([C],HH,H2),
    getGammaId(GId),
    saveGamma(GId,[C,RA,Sol2,H2]),
    Gs:=[GId,G]
  else
    Gs:=[G]
  end;

sigma([C,[],Sol,H],[],cfail,[]):-!;
sigma([C,As,Sol,H],A,C2,AAs):-
  assign(_regAs,As),
  assign(_regGs,[]),
  assign(_regAAs,[]),
  assign(_regCCs,[]),
  repeat,
    _regAs([X|Xs]),
    if delta(C,X,CC0) then
      CC:=CC0
    else
      CC:=cfail
    end,
    append(_regCCs,[CC],_regCCs),
    if CC=cfail then
      xwriteln(0,"constraint violated"),
      append(_regGs,[1.0e+12],_regGs)
    else
    if member(CC,H) or CC=C then
      xwriteln(0,"repeated state"),
      append(_regGs,[1.0e+12],_regGs)
    else
      goal(CC,V),
      append(_regAAs,[X],_regAAs),
      append(_regGs,[V],_regGs)
    end
  end,
  assign(_regAs,Xs),
  Xs=[],!,
  _regGs(Gs),
  if status(trace,-3) then //old version
  xwriteln(0,"[goal(As)  ]=",Gs),stop
end,
  Min:=min(Gs,I),
  if Min=1.0e+12 then
    A:=[],
    C2:=cfail,
    AAs:=[]
  else
    project(As,I+1,A),
    _regCCs(CCs),
    project(CCs,I+1,C2),
    _regAAs(AAs0),
    minus(AAs0,[A],AAs)
  end,
  retract([_regAs,_regGs,_regAAs,_regCCs]);

getGammaId(Id):-
  gammaId(Id),
  assign(gammaId,Id+1);
```

Appendix II extSLV for WGC Problem in extProlog

```
delta(C,A,C2):-
  if A <> f and member(f,C) then
    C20:=minus(C,[f,A])
  end,
  if A <> f and notmember(f,C) then
    C20:=union(C,[f,A])
  end,
  if A = f and member(f,C) then
    C20:=minus(C,[f])
  end,
  if A = f and notmember(f,C) then
    C20:=union(C,[f])
  end,
  sort(C20,C2),
  constraint(C2);

lambda(C,A)=A;
```

```
genA(C,As):-
  if member(f,C) then
    As:=C
  else
    As:=minus([c,f,g,w],C)
  end;

constraint(C):-
  cons(Cs),
  member(f,C),
  notmember(minus([c,f,g,w],C),Cs);
constraint(C):-
  cons(Cs),
  notmember(f,C),
  notmember(C,Cs);

cons([[c,g,w],[g,w],[c,g]]);

initialstate([c,f,g,w]);

finalstate([]);

st(C):-
  finalstate(C);

goal(C,R):-
  R:=strlen(C);
```

```
?-stdPDsolver();


#include "stdPDsolver11.p";
```

Appendix III extSLV for Traveling Salesman Problem in extProlog

```
func("tsp.p",[distance]);
cities([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]);
disList([[0],
            [123,0],
            [56,25,0],
            [78,98,225,0],
            [66,908,140,25,0],
            [37,345,87,678,336,0],
            [999,41,54,402,34,134,0],
            [79,78,89,84,755,62,244,0],
            [30,94,65,44,98,79,745,126,0],
            [74,55,65,110,100,80,96,864,200,0],
            [61,317,58,531,180,39,356,67,46,67,0],
            [88,173,20,970,33,40,678,116,806,19,79,0],
            [138,57,99,340,15,78,10,397,64,76,80,120,0],
            [68,468,15,60,70,198,34,305,79,118,19,369,777,0],
            [105,56,881,576,29,378,169,58,41,60,87,119,12,22,0]]);
delta(C,A,C2):-
```

```
    C2:=append(C,[A]),
    constraint(C2);

genA(C,As):-
    cities(Cities),
    As:=minus(Cities,C);

constraint(C);

initialstate(C):-
    C:=[1];

finalstate(C):-
    cities(Cities),
    strlen(C) = strlen(Cities);

goal(C,R):-
    if (strlen(C) = 1) then
        R:=0
    else
        City1:=project(C,0),
        Vcities:=minus(C,[City1]),
        City2:=project(Vcities,0),
        R:=distance(City1,City2)
    end;

distance(City1,City2,R):-
```

```
    if City1 <= City2 then
        disList(DisList),
        DisCity:=project(DisList,City2),
        R:=project(DisCity,City1)
    else
        disList(DisList),
        DisCity:=project(DisList,City1),
        R:=project(DisCity,City2)
    end;

st(C):-
    finalstate(C);

?-funcCall("tsp.p"),stdPDsolver();
#include "stdPDsolver11.p";
```

Bibliography

[1] D. C. Ince (1988): *An Introduction to Discrete Mathematics and Formal System Specification*, CLARENDON PRESS.

[2] D. Cantone, E. Omodeo and A. Policriti (2001): *Set Theory for Computing*, Springer. [19]

[3] D. Merritt (1984): *Building Expert Systems in Prolog*, Springer.

[4] G. F. Simmons (1963): *Introduction to TOPOLOGY AND MODERN ANALYSYS*, McGRAW-HILL.

[5] J. Bridge (1977): *Beginning Model Theory*, Clarendon Press.

[6] J. Fitzgerald and P. G. Larse (1998): *Modeling Systems*, Cambridge University Press.

[7] M. Covington, D. Nute and A. Vellino (1997): *PROLOG PROGRAMMING IN DEPTH*, Prentice Hall.

[8] M. D. Mesarovic and Y. Takahara (1989): *Abstract Systems Theory*, Springer.

[9] R. Nelson (1968): *Introduction to Automata*, John Wiley & Sons, Inc.

[10] S. Russell and P. Norvig (1995): *Artificial Intelligence - A Modern Approach*, Prentice Hall.

[11] V. Dhar and R. Stein (1997): *SEVEN METHODS FOR TRANSFORMING CORPORATE DATA INTO BUISINESS INTELLIGENCE*, Prentice Hall.

[12] W. F. Clocksin and C. S. Mellish (1987): *Programming in Prolog*, Springer.

[13] Y. Takahara and A. Shimizu (1996): “A foundation of Problem Solving EUD-Conceptual Framework”, J. of Japan Society for Management Information 5 (3.3), (in Japanese).

[14] Y. Takahara and H. Kubota (1989): “Framework for Conceptual Design of Data Processing System”, Office Automation, 10 (1) (in Japanese).

[15] Y. Takahara and T Saito (2003): “Model Building for Problem Solving”, J. of Japan Society for Management Information, Vol.13, No.4, (in Japanese).

[16] Y. Takahara and Y. Liu (1999): “An Extended Prolog for End User Development”, Proc. of World Multiconference on Systems, Cybernetics and Informatics, Orlando.

[17] Y. Takahara and Y. Liu (2005): “Management Information Systems Development: Theoretical Foundation - Model Theory Approach”, Internal report of Chiba Institute of Technology, Japan.

[18] Y. Takahara el al (2005): “System Development Methodology: Transaction Processing System in MGST Approach”, J. of the Japan Society for Management Information, Vol.14, No.1, pp.1-18.

[19] Y. Takahara, T. Asahi and J. Hu (2003): “Application of MGST Design Approach to Data Mining Systems: Case of I-O-O Problem”, J. of the Japan Society for Management Information.

[20] Y. Takahara, Y. Liu and J. Hu (2002): “Intelligent Data Mining System”, Proc. of Int. Conference on E-business, Beijing.

[21] Y. Takahara, Y. Liu, X. Chen and Y. Yano (2005): “Model Theory Approach to Transaction Processing System Development”, Int. J. of General Systems, Vol.3, No.5, pp.537-557.