NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Interactive Hierarchical Generate and Test Search

Xin Xu

Thesis submitted to

the School of Graduate Studies and Research

in partial fulfillment of the requirements for the Master degree in Computer Science

Computer Science Department
University of Ottawa

ISBN 0-315-75026-X

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Acknowledgement

I am extremely grateful to my supervisor Dr. Robert C. Holte who first acquainted me with "Machine Learning", provided me with the initial idea of *Interactive Search*, launched me on this project and helped me with much of it. The contents of Chapter 2 are mainly from him. He also provided the main idea for the generator framework and the generator representation which make all the editing operations possible. Since I became his student, he spared countless hours directing my thesis work including the development of the IHGT search technique, the implementation of the editor GE, and the thesis writing. I learned a lot from him through our weekly discussions. I take this opportunity to express my pleasure in working with him for his wisdom, insight and perspective had profoundly influenced my thinking.

Special thanks are also due to Dr. Stan Szpakowicz who helped me to come and stay in Canada. Thanks also to Dr. Stan Matwin for his kind help.

This work was supported by Dr. Holte from an NSERC operating grant.

Abstract

Most of the search methods used in AI are inflexible. Their search factors (search space, search control strategy, and search goal) cannot be dynamically acquired and changed. Such methods can only be used to solve problems for which the search factors can be determined before search. However, in some cases, search factors are not easily determined. Frequently, it's difficult for a human problem solver to describe what the solutions he is searching for would exactly look like. This kind of difficulty is the result of limitations of human problem solving capacity. For example, in kitchen layout design, goal information is not easy to capture until run time.

Interactive search is a new kind of search in which the search system can communicate and cooperate with *external agents*. There are two kinds of agents: human agents and non-human agents. Through interaction with human agents (*man-machine interaction*), the search system can make use of the human talent of judging the quality of a solution. Through interaction with non-human agents (*machine-machine interaction*), the search system can automatically exploit knowledge from its environment. An interactive search system has the ability to take advice from external agents. The ordinary non-interactive search models are the special instances of interactive search when the advice sequences are empty.

We are investigating a particular kind of *Interactive Search*, IHGT *(Interactive Hierarchical Generate and Test)* search, which is established by introducing *interactive* ability into HGT *(Hierarchical Generate and Test)* search. To make HGT search *interactive*, we created an editor called GE (Generator Editor). GE was implemented in Prolog. GE is a bottom level language shell outside the HGT search model which supports various kinds

of interactions between the HGT search model and its external agents. GE translates advice into dynamic changes of all the three search factors.

GE is suitable for solving real world design problems including routine designs and non-routine designs. The design solutions provided by GE are hierarchical decompositions of design functions. The initial problem solver for a particular design domain is created by assembling components using a priori design constraints. The refinement of the initial problem solver is achieved by taking an agent's advice.

Contents

Chapter 1. Introduction

Most of the search methods used in AI are inflexible. Their search factors (search space, search control strategy, and search goal) cannot be dynamically acquired and changed. Such methods can only be used to solve the problems for which the search factors can be determined before search. However, in some cases, search factors are not easily determined. Frequently, it is difficult for a human problem solver to describe what the solutions he is searching for would exactly look like. This kind of difficulty is the result of limitations of human problem solving capacity. For example, in kitchen layout design, goal information is not easy to capture until run time.

The kitchen layout design problem is to determine the positions of design objects (eg. stoves, tables, etc.) so that all the objects are arranged in the kitchen without overlap. This kind of design problem can be viewed as a search in a very large candidate solution space. The design constraints represent the partial goals for the search model to satisfy. But the difficulty is that the design constraints for kitchen layout cannot be fully given before search. Such constraints can only be determined based on the run-time performance of design solutions. This difficulty can be eased by *interactive search*, which permits the search space, search control strategy and search goals to be dynamically changed.

Interactive search is a new kind of search in which the search system can communicate and cooperate with *external agents*. There are two kinds of agents: human agents and non-human agents. Through interaction with human agents (*man-machine interaction*), the search system can make use of the human talent of solution quality judgement. Through interaction with non-human agents (*machine-machine interaction*), the search system can automatically exploit knowledge from its environment. An interactive search

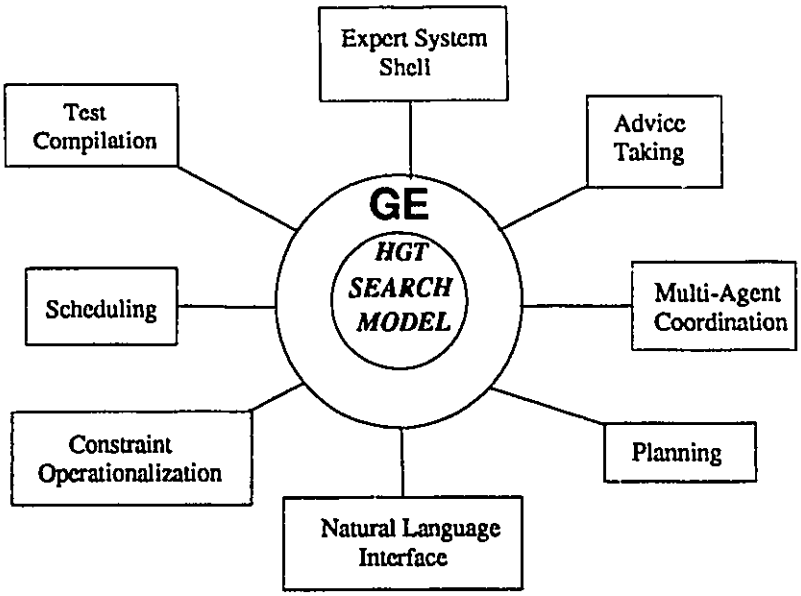
system has the ability to take advice from external agents. The ordinary non-interactive search models are the special instances of interactive search when the advice sequences are empty.

Goals of the research in this thesis are:

- to clarify the functional aspects of the interactive search
- to identify the problems and technical difficulties involved in building an interactive search system
- to find the solutions to the problems
- to build a tool for problem solving based on these solutions

The contribution of this thesis is to make the first attempt in Interactive Search research by investigating a particular kind of interactive search, IHGT *(Interactive Hierarchical Generate and Test)* search. To make HGT *(Hierarchical Generate and Test)* search *interactive*, an editor called GE (Generator Editor) is created. GE was implemented in Prolog. GE is a bottom level language shell outside the HGT search model which supports various kinds of interactions between the HGT search model and its external agents. GE translates advice into dynamic changes of the three search factors.

GE is suitable for solving real world design problems including routine designs and non-routine designs. The design solutions provided by GE are hierarchical decompositions of design functions. An initial problem solver for a particular design domain is created by assembling components using a priori design constraints. The refinement of the initial problem solver is achieved by taking an agent's advice.



As a general-purpose search tool, GE is useful not only for the real world design tasks, but also for AI research. Because GE is a bottom level language shell of HGT, all the AI applications in an HGT environment can be built at different abstract levels by taking GE commands as primitive operators. For example, a natural language interface for HGT search can be built by transforming natural language sentences into GE commands step by step. Likewise, a scheduling problem can be solved by imposing partial or global constraints among subschedulers. Some possible applications of GE are given in the above figure.

In contrast with other heuristic methods, interactive search can acquire knowledge dynamically and the types of knowledge acquired are very diverse. For example, through an agent's advice, interactive search can acquire knowledge of representation (eg. problem structures, etc.), knowledge of changing representations and knowledge of performance assessment.

Our research goals on Interactive Search are *efficiency* and *flexibility* of search methods. *Efficiency* can be improved by incrementally acquiring and incorporating external knowledge. For example, changing control strategy is one way of improving efficiency. GE's flexibility in dynamically changing all three search factors contributes to the flexibility of all the abstract levels above GE. GE provides a set of commands for agents to compose complex changes on search factors. *Change of search factors* and *advice taking* are two of the main issues related to these goals. To reach our goals, search factors need to be dynamically changeable. To make search factors dynamically changeable, external knowledge is needed. Advice taking is our approach to acquire knowledge for changing the search factors. In GE, advice is translated into changes of search factors.

Chapter 2 is a general introduction to the subject of *Interactive Search.*

Chapter 3 investigates a kind of *Interactive Search* called IHGT *(Interactive Hierarchical Generate and Test)* search. IHGT search is developed by improving HGT (Hierarchical Generate and Test) search.

Chapter 4 discusses the design of the editor GE. In 4.1., we describe the representation of the IHGT search model. In 4.2., we discuss some general changes in IHGT search representation including how the generator hierarchies are changed, how states are changed when the hierarchy structure is changed, where the tests should be added, and how the tests should be added. Two kinds of tests (component tests and trigger tests) are discussed. A complete list of GE commands including command syntax and function is given in Appendix A.

Chapter 5 discusses potential applications of GE in the domain of design.

Chapter 6 is a discussion of related work. A comparison with the version-space search demonstrates the flexibility of IHGT search. A comparison with Wile's generator framework shows that our generator framework has very simple structure but very powerful capacity. A comparison with Mostow's work raises issues in advice learning, algorithm representation, and transformation. Comparisons with constraint satisfaction and the reinforcement learning demonstrate the significance of interactive search.

Chapter 2. Interactive Search

2.1. What Is Interactive Search ?

A search method consists of three main factors:

1. search space
2. control strategy
3. search goals

In most search methods, all the search factors are fixed. Such inflexibility restricts the further development of machine intelligence. The reason for the inflexibility is the lack of the ability to exploit *external* knowledge. We define the *external* knowledge provided by external agents as *advice*. The idea of *interactive search* aims at weakening such inflexibility by interactively supplying definitions of the search space, control strategy and goal subspace through *external* advice.

Interactive search is a kind of search which allows the communication and cooperation with external agents.

- There are two kinds of agents, human agents and non-human agents. By interaction with human agents (man-machine interaction), the search system can acquire knowledge from humans incrementally. By interaction with non-human agents (machine-machine interaction), the search system can exploit knowledge from its external environment.
- The *search system's role* is to provide the agent with the context for his advice based on some features of its states which are previously defined and known to the agent.

- The *agent's role* is to supply some initial information about a new domain and then incrementally provide more details based on the state of search system when it is running. The syntax of the advice language is predefined and the meaning of advice is dynamically interpreted in the specific context of the search procedure.
- In the interaction between the search system and the agent, the agent is dominant. The search system itself does little reasoning. It accepts advice from the agent and makes corresponding changes in the search system. The decision making and the evaluation of the search objects are mainly by the agent.
- The search goals might be incomplete and uncertain. Some goal information can be characterized by a set of constraints to be satisfied; some might be in the agent's mind and be transferred into the search system during the interaction. Some might not be in the search system, nor in the agent's mind, but will occur to the agent during search, inspired by the dynamic context. In this way, interactive search supports creative thinking. The goal uncertainty of interactive search partly comes from the uncertainty in the agent's mind. This raises the need for the search system to permit the agent to change his mind, to add, withdraw or modify advice.
- Interactive search is an extension of non-interactive search. The difference between interactive search and non-interactive search is that the search system in interactive search can accept an advice sequence from external agent during search. Non-interactive search is a special instance of interactive search when advice sequence is empty.

There are two main aspects of *interactive search.* They are *change of search factors* and *advice taking.* The principle of *advice taking* in *interactive search* is to make the

advice *operational*, that is to transform the advice into the *changes of search factors*.

2.2. Change of Search Factors

A search space can be changed in many ways depending on the nature of the chosen search method, for example by changing the representation of search model, the structure of the problem to be solved, etc. Successive changes produce a sequence of search spaces Si ($0 \leq i \leq n$). S0 is the initial search space. Sn is the final search space. There are two kinds of primitive changes: *reduction* and *enlargement*. Other changes can be the complex combinations of these two.

Reduction is a change which satisfies: $S_i \supseteq S_{i+1} \quad (0 \leq i \leq n-1)$. *Reduction* narrows the search space. Search by candidate elimination and GT (Generate and Test) are examples of reduction. *Enlargement* is a change which satisfies: $S_i \subseteq S_{i+1} \quad (0 \leq i \leq n-1)$. *Enlargement* broadens the search space. This is needed when a search object is not in the initial search space. The combinations of *reduction* and *enlargement* construct *flexible changes*.

A *search control strategy* determines the search ordering. In interactive search, the search ordering can be dynamically changed by changing the internal search state, or by changing the internal structure of the search model, for example by changing the order of components.

The *goal* information is implied by the search constraints to be satisfied. The constraints can be given before search or during search. The modification of the search goals is mainly carried out by the agent. The agent's advice characterizes partial goal information. *Interactive search* allows agents to change their minds by adding advice or withdrawing or modifying the advice already given.

2.3. Advice Taking

Advice information is an essential kind of external knowledge for obtaining flexible search with changeable search factors. The agent's advice on the current search space affects the subsequent generating sequence by specifying partial characterization of the search goals [Holte,1986].

Advice taking can be viewed as the process of goal acquisition and integration which results in the changes of other search factors. The advice given by external agents implies partial goals to be satisfied. Advice can be translated into many kinds of changes in a search model. Advice taking by changing composition structures of search model is more efficient than simply adding tests [Holte,1986][Holte,1988]. Changing composition structure means to integrate partial goal information into the internal search organization.

The principle of advice taking is to make the advice operational. At the bottom level of the search model, the operationalization of the advice is to transform the advice into the changes on the search factors. The capacity of dynamic advice taking determines the communication capacity of interactive search.

Chapter 3. Interactive Hierarchical Generate and Test Search (IHGT)

We chose HGT (Hierarchical Generate and Test) search as the search model for investigating the interactive search technique. To make HGT interactive, we developed a flexible search technique called IHGT (Interactive Hierarchical Generate and Test) by building a language shell which supports the communication between HGT search system and agents. We designed an editor called GE as such a language shell. In the following, we will describe HGT search, and then we will introduce the basic idea of IHGT search.

3.1. Hierarchical GT Search (HGT)

In order to understand HGT easily, we first discuss GT search, and then extend GT search into HGT search.

A basic GT search system consists of a generator which produces all the possible candidate solutions and a tester which evaluates each candidate. In GT search, the generator defines the candidate space. The tester constrains it by eliminating those candidates that do not satisfy some criteria. In GT search, a domain independent generator can be used to solve domain specific problems by using domain specific constraints. For example, an integer generator can be used to generate odd numbers, even numbers, etc. by adding corresponding constraints.

The main problems of basic GT search are:

- It is difficult to design a generator which can generate *all* the possible solutions, or to determine whether *all* the possible solutions will be generated by an existent generator.

- The generators in basic GT search are non-decomposable. There is no efficiency gained by adding a test to a non-decomposable generator.

One solution to improve the above problems is to introduce hierarchical generator structures. GT search with hierarchical generator structures is called HGT (Hierarchical GT) search. In problem solving by HGT search, hierarchical generator structures correspond to the hierarchical problem structures. Hierarchical problem structure is the functional or structural decomposition of the problem to be solved. Such hierarchical structures permit the effective pruning of candidate solutions by applying constraints early to subclasses in the hierarchy. In HGT, problems to be solved are decomposed into sub-problems, and sub-problems are further decomposed. HGT search has many applications. For example, Flemming used the HGT search for the design of building layouts and the design of residential kitchens [Flemming,1989].

Some observations about HGT:

- efficiency can be improved only when the constrained generator is used as a component.
- efficiency increases as tests are applied earlier in the generator hierarchy.

In HGT search, the search space is determined by a hierarchical problem representation, and the search goals are implied by the constraints imposed in the hierarchy. Like all non-interactive search methods, all three search factors in HGT search are fixed during search. That is:

- problem representation is fixed before search
- search control strategy is fixed before search
- constraints are usually given before search

To make all the search factors dynamically changeable, we developed Interactive HGT (IHGT) which is obtained by adding a language shell GE to the HGT search model.

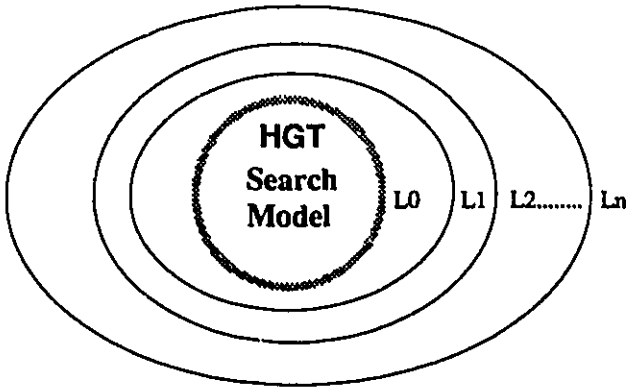
3.2. Interactive HGT (IHGT)

IHGT is a kind of HGT with dynamically changeable search factors. In IHGT, domain independent generators can be used to solve domain specific problems by incrementally supplying domain specific constraints. Problem solving in IHGT includes an initial problem solver and a sequence of advice which results in a sequence of transformations on the problem solver. IHGT allows the initial problem representation to be incomplete, inefficient or even incorrect. The further interaction results in the improvement of the problem solver by acquiring performance judgement, preferences, etc. A priori domain dependent information can be used in building the initial problem solver (possibly very crude). When the initial algorithm can solve the problem correctly and efficiently, no transformations are needed. In such a case, the advice sequence is empty.

The problem representation in IHGT is a hierarchical description. For one problem, there may be many descriptions. For example, a room can be described as walls, ceiling and floor; it can also be described by the coordinates of lower left corner and upper right corner in a Cartesian coordinate system. In IHGT search, the burden of determining the problem representation is greatly eased because the interactive process provides incremental decision making capacity.

The flexibility of changeable search factors in IHGT is provided by the bottom level advice taking facility of our editor GE. IHGT is composed of a kernel HGT search model and a language shell GE. GE translates the agent's advice into a sequence of changes in the HGT search system.

One aim of the GE design is to provide a set of primitive operators (L0) with which to build high level language shells Li ($1 \leq i \leq n$). Ln represents the natural language level.



Advice operationalization [Mostow,1983a,1983b] can be viewed as the transformation of the advice in a given language Li into a sequence of operators at operational level L0. When advice language is in natural language Ln, the advice operationalization can be performed step by step through the intermediate languages Li ($1 \leq i \leq n-1$). This gives us a simple view of how the advice taker near the primitive level can be incrementally extended to accept natural language. Taking GE commands as primitive operators, we can build a complex advice taker which accepts advice language in various complexity levels.

Chapter 4. The Generator Editor (GE)

In this chapter, we describe in detail the editor GE. We first introduce how the IHGT search model is represented, and how such representation is changed. Then, we briefly describe GE commands.

To make the description clear, we use "+", "-" and "?" as the prefixes of arguments. "+" indicates input arguments. "-" indicates output arguments. "?" indicates the arguments which can be input or output.

4.1. Representation of IHGT Search Model

The IHGT search model is represented by generator hierarchies which consist of three kinds of components: generators, tests and functions. The components in the generator hierarchies are connected by composition structures which determine the data flow and control flow among the components. 4.1.1. describes the generator framework underlying GE. This generator framework enables agents to define their own primitive generators and to construct compound generators by composing already existing generators in various kinds of composition structures. 4.1.2. describes how to define each kind of component. 4.1.3. describes seven kinds of composition operators which build the hierarchies. 4.1.4. describes the general structure of a generator hierarchy. 4.1.5. describes the *generator record* which is the representation of a generator hierarchy. 4.1.6. discusses the naming facility in GE which is very important to the success of interactive search.

4.1.1. The Generator Framework

A *Generator* is a procedure characterized by the following factors:

- **para:** parameter of a generator

- **seq**: the sequence of all the states generated by the generator
- **map**: a mapping from parameter **para** to generating sequence **seq** :

$$map: \ para \rightarrow seq$$

In the later descriptions, we will use para(G), seq(G), and map(G) to represent the above three factors of the generator G respectively. We will also use the notation $seq(G)_i$ to represent the ith value in the output sequence of generator G.

Our generator framework is a state transition machine defined by a function called a *Generative Function* (GFN). A GFN defines state transitions based on the current state and the parameter.

Each GFN is represented in Prolog in either of the following forms:

$$G(+P, +S0, -S1).$$

$$G(+P, +S0, -S1) : -GFNBody.$$

G is the generator name. P is the parameter. S0 is the current *generator state*. S1 is the new *generator state*. Each *generator state* is a triple:

$$(PH, GV, IS)$$

- PH: transition PHase
- GV: Generating Value
- IS: Inner State which is the state localized to the GFN

There are three kinds of transition phases for each generator. They are:

- *initial phase*: the phase before generating any value.

- *normal phase*: the phase between initial and exhausted phases.
- *exhausted phase*: the phase after the last value has been generated.

The generating values are part of the states. A GFN produces one value each time a new state is generated. This is called *value stepping*. The value stepping control is mainly from the external agent.

In order to address the issue of how deep we expect the external agents to interfere with the dynamic process, we introduce the concept of inner states. Inner states are used for a generator's local variables. Local variables are the variables used only by the clauses of the corresponding GFN. Some GFNs may need inner state to pass messages between state transitions, some may not use inner states. For example, if the function body uses a pointer to record the position of the current value, the pointer can be stored in the inner state.

The following is a GFN named "enumerate_list" which enumerates elements from a list. The list is the parameter of this GFN.

```
enumerate_list([],(initial,_,_),(exhausted,_,_)).
enumerate_list([H|T],(initial,_,_),(normal,H,T)).
enumerate_list(_,(normal,_,[]),(exhausted,_,_)).
enumerate_list(_,(normal,_,[H|T]),(normal,H,T)).
enumerate_list(_,(exhausted,_,_),(exhausted,_,_)).
```

The first two clauses describe how the first value can be obtained in the "initial" phase. The first clause deals with the case when the input parameter is an empty list in the "initial" phase. Because no value can be generated from an empty list, the new phase is set to "exhausted". The second clause deals with the case when the parameter is a non-empty list in the "initial" phase. The first element H in the list is the value to

be output and the rest of the list T is stored in inner state for the subsequent generation. Because a value H has been generated, the new phase is "normal". The third and the fourth clauses generate next value from a list stored in the inner state. The last clause is used for termination.

4.1.2. Components

IHGT systems are built out of three kinds of components: *generators, functions* and *tests*. Generators are the essential components. Functions and tests are the auxiliary components which modify the competence and the performance of the generators.

Generators

The generators which participate in the compositions are *instance generators*. *Instance generators* are the active generators built from GFNs.

A generator name labels an instance of a GFN. Each GFN defines a generator *type* which implies an infinite number of instances. By giving different parameters to a GFN, different instances of the generator type are obtained. Even the same value of a parameter corresponds to infinite number of instances by different instance labels.

There are two kinds of generators: primitive generators and compound generators. Primitive generators are the minimum generator components. Compound generators are the generators built by composing other components. GFNs are used to define primitive generators. They can also be used to define the composition structures of compound generators (see 4.1.3.).

Functions

A *Function* is any kind of mapping with one input and one output. Each *function* takes either of the following forms in Prolog:

$$F(+I, -O).$$

$$F(+I, -O) : -FuncBody.$$

F is the function name. I is the input argument. O is the output argument.

Functions are useful for adapting a generator to a new usage by modifying the input or the output of a generator.

The following is a simple example defining a function named "add1" which adds one to the input integer.

```
add1(X,Y):- Y is X + 1.
```

Tests

A *Test* has one input and no output. The input either passes or fails the test. Each test has either of the following forms in Prolog:

$$T(+X).$$

$$T(+X) : -TestBody.$$

T is the name of a test. X is the input argument. Applied to the output of a generator, a test filters out all the values which fail it.

Suppose there exists a generator g1 which enumerates all the integers in range [0,100]. If only the integers in the range [0,50] are needed, we don't need to design a new generator specific to the range [0,50]. We need only to attach a test (e.g. t1) to g1 to eliminate the values in range [51,100]:

```
t1(X):- X < 51.
```

4.1.3. Composition Operators

Composition operators create composition structures by connecting components. Composition structures determine the competence and the performance of compound generators. Different composition structures define different data and control flows.

Each composition operator is implemented by a GFN whose parameter is a list of component names. The GFN body defines how the value sequence is generated based on the corresponding data flow control. Composition operators can be defined, modified and deleted by defining, modifying and deleting corresponding GFNs.

In the following, we describe seven commonly used composition operators. They are "infunc", "outfunc", "test", "cascade", "product", "concatenation", and "concurrent". The first four operators create sequential structures. The latter three operators create parallel structures. In the sequential structures, the parameter dependency relations are determined by the sequential connections. In the parallel structures, the parameter dependency relations can be specified by a *parameter binding expression.* Suppose G is a generator with parallel structure which takes Gi ($1 \le i \le n$) as components. The default parameter binding of G is para(G)=[para(G1),para(G2),...,para(Gn)] and para(Gi) are independent each other in this case. This means that G's parameter is used as the parameter of each of its components.

Sometimes it is necessary to give different parameter values to different components of a generator. The relation between each component's parameter and the generator's parameter is defined by giving parameter expressions after generator names in the following form:

[G(E),G1(E1),...,Gn(En)]

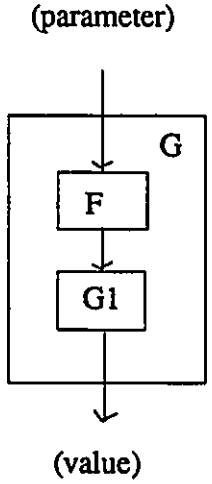
G is the name of the current generator. E is the parameter expression for G. Gi is the name of the ith component generator. Ei is the parameter expression for Gi ($1 \leq i \leq n$).

For example, [g([X,Y]),g1(X-Y),g2(X*Y+1)] means that the parameter of generator g takes the terms of a list [X,Y], the parameter of g1 is the value of expression X-Y and the parameter of generator g2 is the value of the expression X*Y+1.

The parameter binding can be changed freely. The value assignments of the parameters are based on the dynamic interpretation of the parameter binding. Parameter binding is a useful facility for changing the data dependency among the generators.

The "infunc" Operator

The "infunc" operator corresponds to a GFN named "infunc" which takes a function F and a generator G1 as a parameter and generates states by applying F to the input of G1. When an instance generator G is created by this GFN, we have the following data flow:

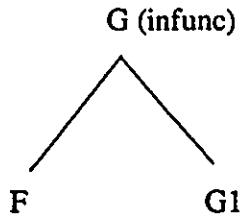


para(G)=input(F)

para(G1)=output(F)

seq(G)=seq(G1)

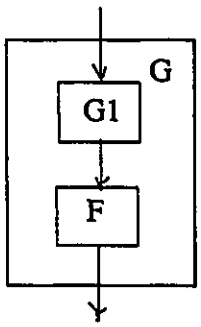
The composition structure is:



The "infunc" structure is used to preprocess a generator's input.

The "outfunc" Operator

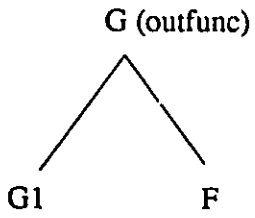
The "outfunc" operator corresponds to a GFN named "outfunc" which takes a generator G1 and a function F as a parameter and generates states by applying F to the output of G1. When an instance generator G is created, we have the following data flow:



para(G)=para(G1)

$seq(G)_i=F(seq(G1)_i)$

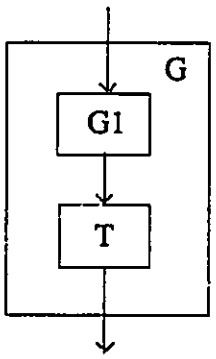
and the following composition structure:



The "outfunc" structure is used to post-process a generator's output.

The "test" Operator

The "test" operator corresponds to a GFN called "test" which takes a generator G1 and a test T as a parameter and generates states by applying T to the output of G1. When an instance generator G is created by this GFN, we have the following data flow:

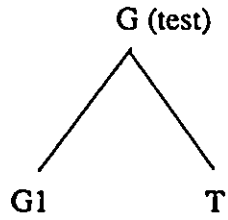


para(G)=para(G1)

seq(G)={ v |∀v, v ∈ seq(G1),T(v) is true }.

G and G1 produce states in the same order.

We have the following composition structure:

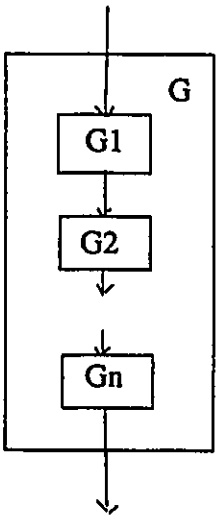


This structure is useful when the filtering of a generator's output is needed.

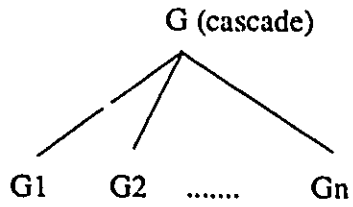
The "cascade" Operator

The "cascade" operator corresponds to a GFN named "cascade" which takes a list of generators Gi (i=1,...,n) as a parameter and uses the output of Gi as the input of Gi+1 ($1 \leq i \leq n - 1$).

When an instance generator G is created with components Gi ($1 \leq i \leq n$), we have the following data flow:



The composition structure is:



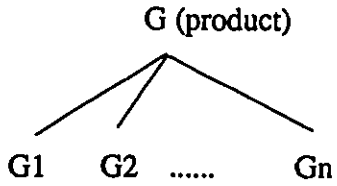
In the "cascade" operation, for each value of Gi, all the values of Gi+1 must be generated ($1 \le i \le n-1$). Gi generates a new value only when Gi+1 becomes exhausted.

The "product" Operator

The "product" operator corresponds to a GFN named "product" which takes a list of generators Gi ($i=1,...,n$) as a parameter and generates the Cartesian product of all seq(Gi) ($i=1,...,n$).

For example, suppose we want to use G1 and G2 to create G in product structure, seq(G1) is [1,2], seq(G2) is [a,b], then seq(G) will be : [[1,a],[1,b],[2,a],[2,b]].

When an instance generator G is created with components Gi, we have the following composition structure:



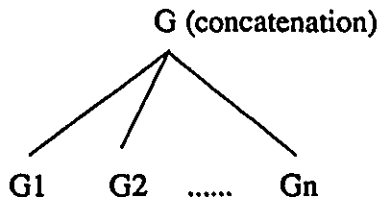
For each value of Gi, all the values of Gi+1 will be generated ($1 \le i \le n-1$). Gi generates a new value only when Gi+1 becomes exhausted.

The "concatenation" Operator

The "concatenation" operator corresponds to a GFN named "concatenation" which takes a list of generators Gi (i=1,...,n) as a parameter and generates:

$$seq(G)=seq(G1)Useq(G2)U,...,Useq(Gn)$$

When the instance generator G is created with components Gi ($1 \le i \le n$), we have the following composition structure:

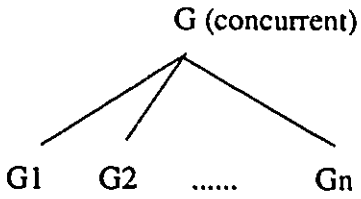


The implementation of concatenation structure takes the first generator as the active generator, the values of the structure are from active generator. When the active generator is exhausted, the next generator in the list becomes active. The structure is exhausted when all the generators in the list are exhausted.

The "concurrent" Operator

The "concurrent" operator corresponds to a GFN named "concurrent" which takes a list of generator Gi (i=1,...,n) as a parameter and generates states by concurrently stepping all the component generators. Suppose v_{ji} represents the jth value of Gi. G is exhausted when there is Gi which is exhausted. The jth value in the output sequence of G is $[v_{j1},v_{j2},...,v_{jn}]$.

When the instance generator G is created by components Gi ($1 \le i \le n$), we have the following composition structure:



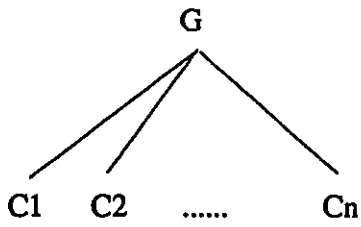
4.1.4. The Generator Hierarchy

The instance generators built by the composition operators are compound generators which have the following type mappings:

Composition Structure	*Type Mapping*
infunc	F * G -> G
outfunc	G * F -> G
test	G * T -> G
product	list of G -> G
cascade	list of G -> G
concatenation	list of G -> G
concurrent	list of G -> G

Here, G means generator, F means function, T means test.

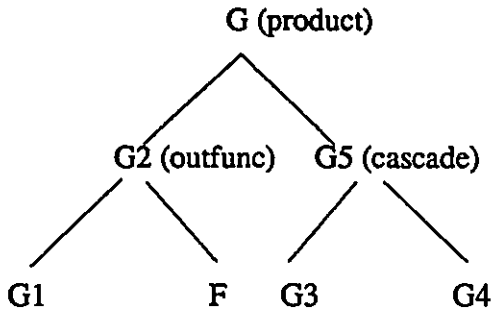
When composing a list of components Ci (i=1,...n) to form a new generator G, the hierarchical relation from G to the components Ci has been determined:



G is the parent of Ci (i=1,...n). Ci (i=1,...,n) are the children of G. The component Ci can be a primitive component (function, test, and primitive generator) or a compound

generator. Compound generators can be further decomposed into smaller components to construct a hierarchy from Ci to its sub-components. In our GE, we introduced the following restrictions to obtain a specific kind of hierarchical structure: no loops and no shared generator names. The compositional structure under such restrictions is actually a tree structure called the *generator hierarchy*.

In the generator hierarchy, the generators which have no parents are called root generators. The generators which have no children are called leaf generators. Leaf generators are primitive generators created directly using GFNs. For example, in the following hierarchy:



generator G2 is created by composing generator G1 and function F in the "outfunc" structure, generator G5 is created by composing generator G3 and G4 in the "cascade" structure, the top generator G is created by composing generator G2 and G5 in the "product" structure. In this hierarchy, generator G1, G3 and G4 are primitive generators, G2 , G5 and G are compound generators.

4.1.5. Generator Records

The representation of generators is essential to the dynamic changes of IHGT search. In GE, we use a *generator record* to represent all the dynamic features of a generator instance. The creation of a generator instance is the creation of the corresponding generator record. The execution of a generator is carried out by the interpretation of the corresponding generator record. Generator records make all the information related to the generators explicit. This enables all the generator operations (eg. composition, decomposition, replace, modification, copy, etc.) to be performed by editing the generator records.

Each *generator record* includes the following information:

$$generator(G, GFN, Para, Comp, Pt, GS, [Names, ParaBond])$$

1. G —Generator name. A generator name labels an instance generator created from a GFN. Each generator has a unique name. The name is given when the instance generator is created.
2. GFN—The name of the GFN from which the instance generator was created. For a compound generator, the GFN name is also the composition structure name.
3. Para — Parameter.
4. Comp—Component list.
5. Pt—Parent generator name in the generator hierarchy.
6. GS—Generator state.
7. Names—Naming list of output value.
8. ParaBond—Parameter binding.

4.1.6. The Naming Facility

Conversation acts require the speakers to refer to individuals, objects, events, etc., in such a way that the listener can mentally identify the referents. The speaker should take into account the listener's knowledge or awareness of a particular object in making reference to that object within a discourse topic [Ochs, 1983]. A natural way of identifying a referent is by naming it. A name identifies the location of the referents. In our IHGT, the internal structures and features of the search system are all visible to the agents. Unlike human communication, the naming information in IHGT has no associated semantic information, but only structural or syntactic information.

The naming and structural information is the basis for constructing an advanced interactive competence. It's a significant effort to build a bottom level language device of a search system using syntactic knowledge of search model itself. The domain-independent nature of syntactic knowledge enables our GE to be a useful tool for various kinds of applications.

Naming facility is an important aspect of any interactive system. The naming facility in GE is:

a. every component(generator, test, function) has a name.
b. any output values or any parts of the output values can have names (optional). The agent can not only assign a name to the whole output value but also assign the names to any entities in the output value.
c. every source of global data or local data can have a name.
d. agents can use names in their advice.

In summary, all the entities which can be involved in the interaction can have names, and the named entities can be referenced in the agent's advice during the interaction.

4.2. Design Considerations

There are many considerations in the design of GE. Dynamic editing is the essential one. In order to make HGT search dynamically editable, some further considerations are: what editing operations are needed (4.2.1.), how to ensure *consistency* when an editing operation is performed in a generator hierarchy (4.2.2.), how an agent's constraints can be dynamically attached into the generator hierarchy (4.2.3.).

4.2.1. Basic Operations on the Generator Hierarchy

Because all the dynamic information about a generator is explicitly represented in the corresponding generator record, we can change any aspects of a generator by changing its generator record.

1. Creating Components: Before creating the generator hierarchy, components need to be created. There are three kinds of components: generators, functions, and tests. Generator components are created by GFNs. GFNs need to be defined before creating the corresponding instances as components. The creation of an instance generator is the creation of the corresponding generator record with the generator name and the GFN name specified. Functions and tests are defined in Prolog.
2. Creating the Hierarchy Structure: A hierarchy can be built by making a copy (see A.10. for "copy" command) or by composing components step by step with specified structures (see A.9. for "compose" command).When composing a list of components with a specified structure, the hierarchical relations from the new generator to the

components will be created by setting the corresponding "parent" and "component" entities.

3. Modifying hierarchy: Components can be attached into a hierarchy, detached from a hierarchy, or replaced by other components (see A.13., A.14., A.17. for "attach", "detach", and "replace" commands).
4. Moving up and down the hierarchy. This is performed by tracing the "parent" and "component" entities in the generator records of the generators in the hierarchy (see A.1. for "look(parent,G,Pt)" and "look(comp, G,Comp)" commands).
5. Viewing all the dynamic data: The ability to view the dynamic data is very important for the agents to obtain the information from search system which directs the bias of the coming interaction (see A.1., A.2., A.3. for "look", "check" and "find" commands).
6. Other changes: The states of generators can be changed by changing the "state" entity in the generator record. When the state of a generator is changed, the states of the surrounding generators need to be changed correspondingly. Similarly, when other aspects of a generator are changed, changes in other generators might be needed based on the dependency in generator hierarchy. Surrounding state change is the subject of the next section.

4.2.2. Changing a Generator's Surrounding State

Changes in one generator might require changes in other surrounding generators. The main surrounding changes include changes of the surrounding hierarchical relations and changes of the surrounding states. For lower level commands (see 4.3. level 1 commands), the change of surrounding hierarchical relations is the responsibility of the

agent. For the higher level commands (see 4.3. level 2 commands), the change of surrounding hierarchical relations is performed automatically.

In the following, we describe the local surrounding state changes initiated by local changes in the generator hierarchy. The changes needed to produce global consistency are performed automatically by the value stepping ("next") command.

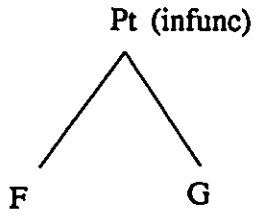
Surrounding states of generator G include :

- the states of G's parent
- the states of G's brothers

G's brothers are the components which have the same parent as G.

Suppose PH(G) represents the transition phase of generator G and GR(G) represents the output value of generator G. The surrounding state of G is changed by command "modify(s-state,G)".

1. in the **"infunc"** structure

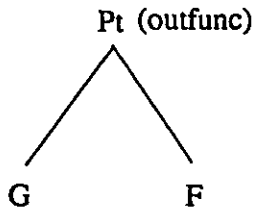


When the state of generator G is changed, the state of Pt will have the following change:

$$PH(Pt) \leftarrow PH(G)$$

$$GR(Pt) \leftarrow GR(G)$$

2. in the **"outfunc"** structure

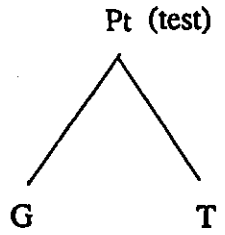


When the state of G changes, the state of Pt will have the following change:

$$PH(Pt) \leftarrow PH(G)$$

$$GR(Pt) \leftarrow F(GR(G))$$

3. in the **"test"** structure



When the state of G is changed, the state of Pt will have the following change:

if $T(GR(G))$ is true, then

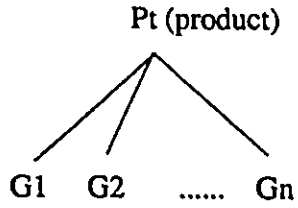
$$PH(Pt) \leftarrow PH(G)$$

$$GR(Pt) \leftarrow GR(G)$$

else, state of Pt remains unchanged.

The change will be done by the next value stepp.ng of Pt. After this stepping, the state of G and Pt will be the same.

4. in the **"product"** structure



Suppose

$$GR(Pt) = \left[V_1, ..., V_{i-1}, V_i, V_{i+1}, ..., V_n\right]$$

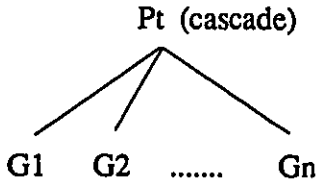
When the state of Gi ($1 \leq i \leq n$) is changed, the state of Pt will have the following change:

if $PH(Gi) = normal$, then

$$GR(Pt) \leftarrow \left[V_1, ..., V_{i-1}, GR(Gi), V_{i+1}, ..., V_n\right]$$

else, the state of Pt will be dynamically changed by the next value stepping on Pt.

5. in the **"cascade"** structure



Suppose

$$GR(Pt) = \left[V_1, ..., V_{i-1}, V_i, V_{i+1}, ..., V_n\right]$$

When the state of Gi ($1 \leq i \leq n$) changes, the state of Pt will be changed as follow:

$$GR(Pt) \leftarrow \left[V_1, ..., V_{i-1}, GR(Gi), V_{i+1}, ..., V_n\right]$$

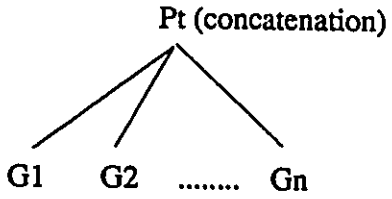
The states of Gi's brothers will be changed as follow:

$$PH(Gj) \leftarrow exhausted$$

$$(i < j \leq n)$$

The reason for doing this is to cause the change of $V_{i+1},...,V_n$ dynamically by the next value stepping on Pt.

6. in the **"concatenation"** structure



When the state of Gi ($1 \leq i \leq n$) is changed, if PH(Gi) is not "exhausted", then the state of Pt will be changed as follow:

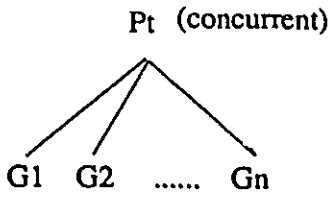
$$PH(Pt) \leftarrow PH(Gi)$$

$$GR(Pt) \leftarrow GR(Gi)$$

The brother state changes are as follow:

1. find the first non-exhausted generator Gj ($1 \leq j < i$)
2. swap the position of Gi and Gj if Gj exists so that Gi will become the active component in concatenation structure

7. in the "**concurrent**" structure



Suppose

$$GR(Pt) = \left[V_1, ..., V_{i-1}, V_i, V_{i+1}, ..., V_n\right]$$

When the state of Gi $(1 \leq i \leq n)$ is changed, the state of Pt will have the following change:

if $PH(Gi)$ = normal, then

$$GR(Pt) \leftarrow \left[V_1, ..., V_{i-1}, GR(Gi), V_{i+1}, ..., V_n\right]$$

else $PH(Pt) \leftarrow exhausted$

4.2.3. Dynamic Test Attachment

Test attachment addresses the questions of how and where a test should be attached in the problem representation.

We recognize two ways of attaching tests: *test appending* and *test incorporation*. *Test appending* involves using a test to filter the values produced by a generator. *Test incorporation* involves changing a generator's micro-structure (code) so that it only produces values that pass the test.

Test appending results in the changes on the composition structure, or macro-change. Changing the location of a test belongs to the *test appending* class. *Test appending* can be performed by a syntactical approach without using semantic information.

Test incorporation results in changes in the non-decomposable components, or micro-change. *Test incorporation* needs to access semantic information. One way to do test incorporation is to provide a set of transformation rules associated with the target component, as is done in Mostow's transformational derivation method.

Dynamic test attachment is test attachment during search. After the user's advice is translated into operational tests, the next step is to apply the tests to a generator by test attachment.

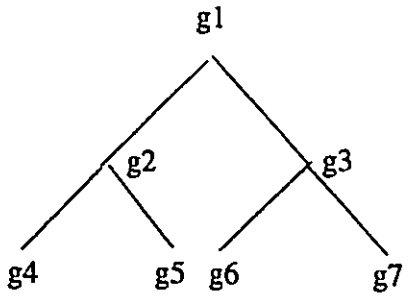
There are many issues surrounding the dynamic test attachment. In the following, we discuss two of them. They are dynamic test location (4.2.3.1.) and dynamic test appending (4.2.3.2.). The issue of dynamic test incorporation is not considered in this thesis.

4.2.3.1. Dynamic Test Location

In IHGT, search efficiency is heavily influenced by the positions where the constraints are attached in the hierarchy. One idea to improve the efficiency is to move constraints as local as possible.

In the composition hierarchy, the *target component* is the position to attach a test. The best position is the *Minimal Generalization (MG)* , the lowest point in the hierarchy tree that covers all the names mentioned in a test.

Consider the following hierarchy:



Suppose mg(G1,G2) represents the minimal generalization of G1 and G2. We have:

mg(g4,g5)=g2

mg(g4,g2)=g2

mg(g4,g3)=g1

mg(g2,g2)=g2

mg(g4,g6)=g1

mg(g6,g1)=g1

When a constraint is dynamically given, the corresponding target component will be determined based on the dynamic search context. Constraint factoring is the process of factoring the constraint in a given language into a constraint meaningful in the dynamic search context of a specific search system.

4.2.3.2. Dynamic Test Appending

Dynamic test appending is an important issue in our GE design. In the following, we discuss two main problems related to this issue and give the solution to the problems.

The Parameter Mismatch Problem

Some problems of HGT are the constraint factoring problem [Braudaway,1989b] and the partial solution evaluation problem [Rolston,1988]. Parameter mismatch is one cause

of these problems. When trying to attach a test in the generator hierarchy, a parameter mismatch occurs when the agent's constraint can't be translated into a test representation in which the input structure of the constraint is consistent with the output structure of the associated generator. In order to attach a test to a generator as in the "test" structure (see 4.1.3.), the input structure of the test must exactly match the output structure of the generator.

Partial Matched Parameter (PMP) is one kind of parameter mismatch. PMP happens when the variables in the constraint only match part of the output structure of the corresponding minimal generalization (MG).

For example, suppose we have a generator g, whose output is a two item list, the items being named "item1" and "item2". In the constraint, $item1 > 10, the name "item1" refers only to part of the g's values. The partially matched parameter problem arises when the generator g can't be decomposed into individual generators among which there is a generator which produces the value named "item1".

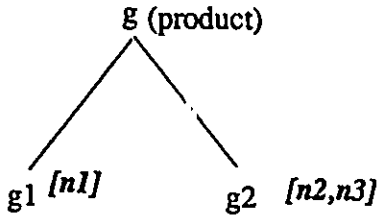
For example, if g is a primitive generator, the above constraint can't be directly attached to g.

The Naming Trouble in Test Attachment

The name information we allowed in our system is only for the communication between the search system and agents, particularly for the component and value references. Name information does not participate in the system's internal reasoning because we do not consider the semantic relationships among names and the semantic associations of each name in our syntactic approach.

The method of test appending already discussed is by data flow connection (see 4.1.3. for test structure), called component test appending. In this kind of appending, the parent generator in a hierarchical generator structure masks its children's naming information. The masked names are invisible from the output data of a target generator. We call this "name masking effect".

The name masking effect occurs when the children's name information is not accessible in the parent's output value, but is referred to by the constraint which is to be attached to the parent generator. Suppose we have a generator structure:



Suppose the value name of g1 is n1 and the value name of g2 is a name list [n2,n3]. When the user's advice refers to the entities in the output of g which correspond to g1 and g2, and g is the target generator of the constraint, the name masking effect arises in the following situation:

a. g has no names for its output value
b. the children's name information does not explicitly appear in parent generator
c. the entity names in the output of g are different from the output value names of the generators generating these entities

Taking g as a target generator, any constraints in which the names are not explicitly available in the output of g can't be attached to g directly as component tests. For example, for the following constraint: $n1 + $n2 * 10 < $n3, the Minimal Generalization (MG) of this constraint is g. This means the constraint must be attached to generator g. When not all the name information about $n1,$n2,$n3 is explicit in the output of g, the constraint can't be attached to generator g as a component test.

Discussion

By investigating the problems mentioned above, we found that the main cause of the problems is the method of adding a test to a generator. The troubles arise because we add a test only by data flow connection.

Data flow control is the essential aspect of generator design and use. A group of generators can be composed by appropriate data flow connection among them. Data flow connection determines the composition, data flow and the input data (parameter) of generators.

The problem is, how to add a test directly to a generator without using a data flow connection. In the following, we introduce a trigger test technique which solves all the above problems in our IHGT search system by avoiding data flow test connections.

Trigger Tests

A *Trigger test* is a kind of test in which the association between a generator and test is based on trigger setting instead of data flow association.

When a generator is value-stepped, all the trigger tests connected to the generator will be checked one by one. An output value must pass all the trigger tests in order to be sent out. If one trigger expression fails the test, the value will be rejected.

A trigger test can use names to directly refer to any components, any parts of the generating values, any generator states or even any local or global data.

The basic model of this technique is a table called trigger-table. Trigger table is a set of (MG,Expr,Expr'). MG is the name of minimal generalization generator of the trigger expression Expr. Expr' is the transformed form of Expr. There are two steps for test appending using the trigger test technique:

1. trigger setting
2. trigger test evaluation

There are the following steps for trigger setting:

1. determine MG
2. transform Expr to Expr'
3. put (MG,Expr,Expr') into trigger table

The first two steps are performed by a grammar directed parser with operational attribute processing rules incorporated into each grammar rule. The parser top-down analyzes the trigger expression, and bottom-up determines MG and transforms the trigger expression into a simple form Expr'. The difference between the input trigger expression and the transformed expression is that the transformed expression includes attribute information for each item, for example, name types, etc. as described in the following.

A generator name item $Gen in the user's trigger expression will be transformed to: "g:Gen". "g" tells the evaluator that the name Gen is a generator name and the value of this item is the current result of generator Gen.

A value name item $F in user's trigger expression will be transformed to: "f:Gen:F" which tells the evaluator that F is a feature name. The target generator of this feature is Gen. Gen is determined during the analysis of the trigger expression. The interpretation of this item is done by locating the sub-item of generator Gen's current result which corresponds to the name F.

The constant Value in the input will be transformed to: "c:Value". "c" means the "Value" is a constant.

For example, if we have the following generators:

```
generator(g1,gen_int,[1,10],[],g3,(non_exhausted,6,_),[_,[]]).
generator(g2,gen_int,[20,30],[],g3,(non_exhausted,20,_),[_,[]]).
generator(g3,product,_,[g1,g2],nil,(non_exhausted,[6,20],_)[[n1,n2],[]]).
```

the expression : $n1 + $n2 * 3 (see A.18.) will be translated as:

```
f:g1:n1 + f:g2:n2 * c:3
```

By input this expression to evaluator, the value is 66: (6+20*3)

The algorithm of the evaluator is also grammar rule directed with semantic processing incorporated into each grammar rule.

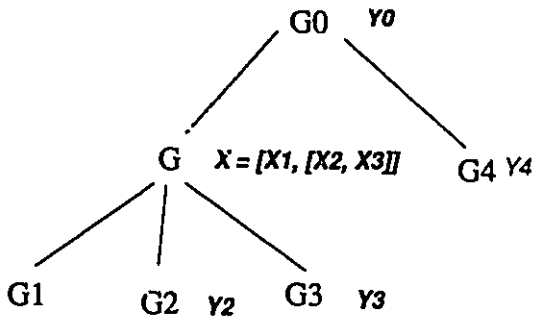
Component Tests and Trigger Tests Compared

There are two ways to add a test in our IHGT search system:

1. add the test as component
2. add the test as trigger

The former can be used for local complete tests. The latter can be used for global tests and local tests whether the test is complete or partial. The differences between a component test and a trigger test are the following:

1. a component test requires that the input structure of the test be consistent with the output structure of the target generator. A trigger test has no such requirement.
2. a component test is an operational procedure, a trigger test is data which needs to be interpreted dynamically.



Suppose Y0 is the output of G0. X is the output of G. Y2 is the output of G2. Y3 is the output of G3. Test T is associated with generator G.

Tests	*Component Test*	*Trigger Test*	
T(X)	OK	OK	Complete, Local Test
T(X1)	NO	OK	Partial Test
T(X2)	NO	OK	
T([X1,X2])	NO	OK	
T(Y0)	NO	OK	Global Test
T(Y2)	NO	OK	
T(Y3)	NO	OK	
T(Y4)	NO	OK	
Y0+Y2*Y3 > X3	NO	OK	

All the dynamic test appending problems are solved by using the trigger test technique. Trigger tests make the transformation and incorporation of agent's elimination constraints easier. Because a trigger test is data, it is easy to make inference, transformation, and optimization on it.

The reasons we keep component tests are:

- Component tests are defined by Prolog, but trigger tests are defined in a very restricted language in the current version of GE.
- Component tests can be used to design an initial search model. The trigger tests in our current GE can only be used for the dynamic performance modification of a search model after it is built.
- Component tests can be shared by many generators. Trigger tests can't be shared because appending a trigger test to a target generator is based on the dynamic context of the target generator.

4.3. Summary of GE's Commands

The advice language for GE is its commands. To make GE a general purpose language shell for HGT search, the choice of the primitive operators is essential. The principal requirements determining the primitive set in GE are:

- keep all the operators as low level as possible
- keep the size of the set as small as possible
- the primitive set must be complete so that any kinds of high level operations can be expressed as the combinations of the operators in the primitive set
- the primitive set must balance the needs of both human and non-human agents

<table>
<tr><td rowspan="3">1</td><td rowspan="3">Look
Check</td><td>Build
Set
Modify
Remove</td></tr>
<tr><td rowspan="4">Find
Statistics</td><td>Compose
Copy</td></tr>
<tr><td>Next
Reset</td></tr>
<tr><td rowspan="2">2</td><td>Attach
Detach
Insert
Extract
Replace</td></tr>
<tr><td>3</td><td colspan="2">Taking Advice Expression</td></tr>
</table>

In GE, the commands occur at three abstract levels: low level, middle level and high level. The main primitive operations in GE are listed in the above table. In this table, the commands in the left column are for the system to provide context for external agents, the commands in the right column are for external agents to build, modify or use the generator hierarchy. Commands in level 1 (low level) perform local changes without considering local or global consistency. Commands in level 2 (middle level) ensure local and global consistency automatically. Commands in level 3 are high level commands.

The agents' advice (GE commands) is dynamically interpreted based on the current context of HGT search procedure.

Details of the GE commands are given in Appendix A.

Chapter 5. Applications

5.1. Real World Design Tasks

As the propelling force of evolution, *design* has been a prevailing task throughout human history. In today's modern world, *design* tasks become even more pressing and comprehensive which make the *design* problem hard to manage in the stereotyped methods. A resort to AI techniques promises to ease the modern design dilemma.

The following summarizes three design models underpinning many intelligent design systems:

(1). E^3 model (Enumeration&Evaluation&Elimination)— Enumerates all the possible candidate solutions and determine the target set of solutions by eliminating the undesired ones through the observation or estimation of design performance.

(2). Template Model —Performs routine design by the different uses of a fixed template.

(3). Composition Model —Performs routine design or creative design by assembling components.

All these design models can be simulated by search. For example, GT search fits the E^3 model. HGT fits the Composition Model. The tailorability of a generator for different applications by parameterization indicates that GT and HGT also fit the Template Model. Viewing design as search provides a simple way to understand and model real world design problems.

The LOOS system [Flemming,1989] and the Press Lineup Advisor system [Lan,1990] demonstrate the GT search technique for real world design tasks. LOOS is an expert

system for the design of building layout. Press Lineup Advisor is a knowledge-based system for the design of newspaper printing press configurations.

5.2. Interactive Design

There are two types of design tools: automatic and semi-automatic. Automatic design requires all design constraints to be given before design. AI research has proven the difficulty of acquiring domain knowledge for expert systems. Automatic design can only be used for some specific problems in which the domain knowledge is not hard to capture. Semi-automatic design is more attractive because human interaction plays an important role in easing the difficulties of knowledge acquisition.

Many real world design activities can be eased by *interactive* method. Man-Machine interaction proved to be very useful for human designers to find design solutions effectively and for computer systems to acquire human expertise easily.

In interactive design, "design" can begin with little domain knowledge. The designed product can be revised and refined dynamically. Design constraints can be given any time during the design process. The interactive process provides an approach to acquire human knowledge. This is very important for the system to evolve towards an automatic system.

Interactive design is the dynamic transformation of a design representation. In GE, the agent selects the transformation, the editor accepts the selection and performs the actual transformation by dynamic interpretation.

The agent plays a prime role in the interactive process. The agent can change his mind at any time based on the performance of a design. The agent can also abandon any part of the design. The agent's decision is based on the context of the dynamic performance of a design. The context for the agent's interaction is mainly from value

stepping (see "next" command), state display(see "look" command) and feature checking (see "check" command) which provide the following:

- a window for the agent to observe the system performance
- a context to affect the bias of interaction
- design candidate display for the agent to evaluate and criticize selection

GE supports two kinds of interactive design: *program design* and *object design* which will be described below.

5.2.1. Program Design

Program design is a kind of assembly process for software development. In "program assembly" one takes already existing programs as components, and composes them in the desired structure. The programs designed by GE are generators with many kinds of internal composition structures.

Program design is the interactive transformation of a design representation. The transformation sequence is from agent's commands. The initial design is created by defining primitive generators. The desired design representation is obtained by a sequence of transformations on the initial design.

In program design, the editing procedure implies a search for a desired program construction. Underpinning the program search, is the issue of *change of representation.* A sequence of editing events result in a sequence of changes in the program representation.

To perform the program design, agents need to build their own GFNs, and create primitive generators from GFNs by the "build" command. After primitive generators are built, compound generators can be built by the "compose" command. In the "compose"

command, agents need to specify the composition structure and the components. After the initial program is established, agents can edit the program structure by any of the following commands: "copy", "remove", "attach", "detach", "insert", "extract", "modify", "replace", "look", "find", "check", etc.

5.2.2. Object Design

Object design follows program design. Object design searches the desired objects by eliminating undesired ones based on the agent's elimination advice. The elimination can be carried out by attaching tests or modifying the generator's composition structure.

The main facilities for object design are value stepping (see "next" command), resetting (see "reset" command) and advice taking (see "advice" command).

Actually, program editing and object search are not strictly separate. Sometimes both kinds of design are mixed together.

5.3. A Simple Example

In this section, we will show the application of GE in the domain of kitchen layout design. To make the description short and complete, we simplify the two-dimension layout problem to a one-dimension layout problem.

One-dimension layout problem:

> Given a set of objects a_i ($1 \le i \le n$) and corresponding widths w_i ($1 \le i \le n$). The problem is to determine the position p_i ($1 \le i \le n$) of each object so that all the objects are arranged on the line without overlap and satisfying a set of constraints.

Suppose each object is defined by the following:

object(ObjectName,FeatureName,FeatureValue).

To make things simple, we assume that each object has only one feature, width, whose value is given by a range of integers. For example,

object(stove,width,[1,2])

means that the width of the stove is in the range [1,2].

In the following, we describe a method of solving this layout problem by designing an initial algorithm and then adding constraints to improve it.

The initial algorithm of one-dimension problem has four steps:

1. design the order of the objects by enumerating all the possible permutations.
2. design the width of each object by enumerating all the possible widths of each object.
3. add a length constraint so that the sum of all the object widths is not greater than the maximum length of the line (wall).
4. design the position of each object by enumerating all the possible plans.

   In one-dimension layout, the position of one object affects the position of the following objects on the line.

   Suppose $[p_{i1},p_{i2}]$ represents the range of position of object a_i, and w_i and p_i represent the width and the position of object a_i. L is the length of the line. The range of p_i is determined by the following algorithm:

$$p_{i1} = p_{i-1} + w_{i-1}$$

$$p_{i2} = L - \sum_{j=i}^{n} w_j$$

$$p_0 = 0, \quad w_0 = 0$$

$$(1 \leq i \leq n)$$

The following is the main program of the above algorithm composed by GE commands:

```
design(Name,LEN,OBJS):- /* design(?Name,+LEN,+OBJS) */
        set(constant,wall_length,LEN),
        build(gen,obj_perm,OBJS,G1),
        build(gen,width_design,_,G2),
        compose(cascade,[G1,G2],G3),
        compose(test,[G3,widthtest],G4),
        build(gen,position_design,_,G5),
        compose(cascade,[G4,G5],Name).
```

"Name" is the name of the initial algorithm given by the agent. "LEN" is the length of the line. "OBJS" is the list of objects participating in the design. "obj_perm" is a GFN to generate all the possible permutations of a list of objects. We do not give the complete definition of this GFN because of its complexity. "width_design" is a GFN for generating all the possible widths of the objects. "position_design" is a GFN for generating all the possible positions of the objects. The algorithm builds the generator G1 with GFN "obj_perm", and the generator G2 with GFN "width_design". G1 and G2 are composed together in a "cascade" structure to form generator G3. "widthtest" is a component test. This test is attached to G3 to form G4. G5 is built with GFN "position_design". Finally, the whole algorithm is completed by composing G4 and G5 in a "cascade" structure.

The following is the definition of GFN "width_design":

```
width_design(PERMS,(initial,_,_),(Ex,R,G)):-
        widthGen(PERMS,Glist),
        compose(product,Glist,G),
        next(G,Ex,R).
width_design(_,(exhausted,_,_),(exhausted,_,_)).
width_design(_PERMS,(normal,_,G),(Ex,R,G)):- next(G,Ex,R).
```

In the above GFN, the "compose" and "next" commands are used. The GE commands can not only be used to do the editing on generator hierarchies but also to define GFNs.

"widthGen" builds a list of width generators corresponding to each object.

```
widthGen([],[]).
widthGen([A|T1],[G|T2]):- object(A,width,B),
        build(gen,gen_int,B,G), widthGen(T1,T2).
```

"widthtest" is a component test which restricts the sum of the object widths to be within the length of the line.

```
widthtest([_PERMS,Widths]):- sumlist(Widths,Sum),
        look(constant,wall_length,LEN), Sum =< LEN.
```

"position_design" is a GFN for position design of each object.

```
position_design([_PERMS,Widths],(initial,_,_),(Ex,R,Name)):-
        length(Widths,N), build(gens,gen_int,N,[G|Grest]),
        look(constant,wall_length,Len), sumlist(Widths,Sum),
        P2 is Len - Sum,
        build(gens,gen_pos,N,[PosG|PosGrest]),
        modify(para,PosG,[_,(0,P2),[G|Grest],Widths]),
        compose(cascade,[PosG|PosGrest],Name),
        next(Name,Ex,R0), pos_output_mapping(R0,R).
position_design(_,(exhausted,_,_),(exhausted,_,_)).
position_design(_,(normal,_,G),(Ex,R,G)):-
        next(G,Ex,R0), pos_output_mapping(R0,R).
```

In this GFN, another GFN named "gen_pos" is used. This illustrates the embedded GFN design.

```
gen_pos([_,_,[],[]],_,(exhausted,_,_)).
gen_pos(_,(exhausted,_,_),(exhausted,_,_)).
gen_pos([_,(P1,P2),[G1|Grest],[W1|Wrest]],(E,_,_),(Ex,R,_)):-
        (E == initial -> modify(para,G1,[P1,P2])
        |otherwise -> true), next(G1,Ex,R1),
        (Ex == exhausted -> true
        |otherwise ->
        sumlist(Wrest,Wsum), look(constant,wall_length,L),
        P11 is R1 + W1, P22 is L - Wsum,
        R=[R1,(P11,P22),Grest,Wrest]).
pos_output_mapping([],[]).
pos_output_mapping([[Pos,_,_,_]|T1],[Pos|T2]):-
        pos_output_mapping(T1,T2).
object(s,width,[1,2]).
object(c,width,[1,4]).
object(f,width,[1,2]).
```

In the following, the "design" program is called to create the initial algorithm named g for designing the one-dimension layout with line length 3 and object list [s,c,f].

```
| ?- design(g,3,[s,c,f]).
yes
```

After the algorithm is created, it might need to be edited to make it correct and efficient. By using "look", "find" and "check" commands, we can inspect the whole hierarchical structure of the algorithm and all other information related to this algorithm. The algorithm structure can be easily changed at any time after the algorithm is created. First, we assume that the algorithm is already correct and efficient. Under this assumption,

we ignore the algorithm transformations and directly go to the "object" search by using this algorithm. The following are some value stepping results (the structure of the output value is [[ObjectList,WidthList],PositionList]).

```
| ?- next(g,E,R).
E =normal,
R = [[[s,c,f],[1,1,1]],[0,1,2]]
| ?- next(g,E,R).
E =normal,
R = [[[s,f,c],[1,1,1]],[0,1,2]]
| ?- next(g,E,R).
E =normal,
R = [[[c,f,s],[1,1,1]],[0,1,2]]
| ?- next(g,E,R).
E =normal,
R = [[[f,c,s],[1,1,1]],[0,1,2]]
| ?- next(g,E,R).
E =normal,
R = [[[f,s,c],[1,1,1]],[0,1,2]]
| ?- next(g,E,R).
E =normal,
R = [[[c,s,f],[1,1,1]],[0,1,2]]
| ?- next(g,E,R).
E = exhausted,
R = _3
```

Now, we show how elimination advice constrains the candidate space.

```
| ?- design(g2,6,[c,s]).
yes
```

This creates a one-line layout algorithm with line length 6 and object list [c,s].

```
| ?- next(g2,E,R).
E =normal,
R = [[[c,s],[1,1]],[0,1]]
```

This is the first value stepping result. The object "c" has the width "1" and the position "0". The object "s" has the width "1" and the position "1".

```
| ?- next(g2,E,R).
E =normal,
R = [[[c,s],[1,1]],[0,2]]
```

In this value stepping result, the position of object "s" is changed from "1" to "2".

```
| ?- next(g2,E,R).
E =normal,
R = [[[c,s],[1,1]],[0,3]]
```

The design of the object widths remains the same until the design of the object positions on this width design is exhausted. If we want to go to some specific design of object widths, we can skip all the value sequence before this design with the "advice" command. Before using the "advice" command, we need to make sure that all the necessary name information is given.

```
| ?- set(name,g2,[[[a1,a2],[w1,w2]],[p1,p2]]).
yes
```

This sets the name "a1" and "a2" to the objects in the output object list, sets the name "w1" and "w2" to the values in the output width list, and sets the name "p1" and "p2" to the values in the output position list.

```
| ?- advice(g2,[$,w1,=,3]).
yes
```

This inserts the constraint "w1 = 3" into the current generator hierarchy rooted at g2. The following is the sequence of value stepping results immediately after the above constraint is given.

```
| ?- next(g2,E,R).
E =normal,
R = [[[c,s],[3,1]],[0,3]]
| ?- next(g2,E,R).
E =normal,
R = [[[c,s],[3,1]],[0,4]]
| ?- next(g2,E,R).
E =normal,
R = [[[c,s],[3,1]],[0,5]]
```

The following is the example of a constraint on the position value and the subsequent value stepping results.

```
| ?- advice(g2,[$,p1,>,1]).
yes
| ?- next(g2,E,R).
E =normal,
R = [[[c,s],[3,1]],[2,5]]
| ?- next(g2,E,R).
E = exhausted,
R = _5
```

Chapter 6. Related Work

6.1. Version-Space Search

The version-space search [Mitchell,1983] uses two boundary sets (the S set and the G set) to represent the search space. S set is the set of most specific hypotheses. G set is the set of most general hypotheses. When a positive example is given, the method covers this example by transforming the S set into more general descriptions. When a negative example is given, the method excludes this example by transforming G into a more specific version. The candidate space is implicitly defined by the S-G boundary. As a result, the candidate elimination procedure involves the tightening of S-G boundary. When the S-G boundary sets become closer, the candidate space becomes smaller.

The similarities between version-space search and IHGT search are:

- Both have the ability to change the search space dynamically. Both can be used to solve problems with goal-uncertainty. The S set and the G set are boundary sets which restrict the uncertainty to the space between them. The boundary sets are dynamically changeable and the changing direction is toward more certainty.
- Both can dynamically acquire human knowledge by allowing human to participate in the interaction or decision making. The version-space method can do so by learning examples from human. The IHGT search method can do so by learning advice from human.

The differences between version-space and IHGT are the following:

Version Space	*IHGT*
search space is represented by S-G boundary sets	search space is represented by a hierarchical structure
user's information is only about classification information (negative, positive examples)	user's information can be more than classification information, such as an assessment of the system's performance, constraints to be satisfied, etc.
only results in a reduction of the search space	the search space can be any combination of reduction and enlargement

It's not clear how easy and how well a boundary representation can be used for real world problem solving, but there's no doubt that a hierarchical structure is powerful for modeling real world problems in various kinds of domains.

6.2. Wile's Generator Framework

At a conceptual level, our generator framework is similar to Wile's generator framework. Wile's generator framework consists of four functions: [Wile,1982]

1. *Decode State Function* which determines what constructs the generated "element"
2. *Initial State Function* which produces the initial state from the empty state
3. *Next State Function* which produces a new state from a previous state
4. *Terminal State Predicate* which determines whether the generator has run out of elements.

In our generator framework, each GFN must give a complete description of the state transitions for all the three phases (initial, normal and exhausted). The transition definition in the "initial" phase is similar to the "Initial State Function". The transition definition for the "normal" phase is similar to the "Next State Function". The transition definition for the "exhausted" phase corresponds to the "Terminal State Predicate".

Wile's generator expressions provide an easy and precise semantic description of loops and yield programs which are easy to be compiled, interpreted or transformed. Our GE can also be used to describe loops in the IHGT environment.

The differences between Wile's generator framework and our generator framework are listed in the following table.

Wile's	*IHGT*
generator definition is declarative	generator definition is procedural
can refer to generators by name	can refer to any component by name
tests can be used to filter values . e.g. "when" operator. Tests can also be used to control loop termination.	tests can only be used for value filtering
some composition operators involve three kinds of components (generator,test, function) at the same time. e.g. "finally", "afterwards" controls generator-test-function sequence	no composition operators concern the relationship between tests and functions.
suitable as a looping facility in a programming language, but not suitable for general purpose generator design	suitable for general purpose functional software development in software engineering

6.3. Mostow's work

6.3.1. Advice Learning

Advice learning is a typical model of learning by being told. The key problem in learning by being told is *operationalization* [Mostow,1983a]. To make advice operational, Mostow's approach is to transform advice into a heuristic search procedure [Mostow,1983a].

> "The derivation consists of a series of problem transformations leading from the advice statement to an executable procedure. The operators used to perform these transformations are implemented in a program called FOO as domain-independent transformation rules that access a knowledge base of task domain concepts. Some of the rules construct a crude generate-and-test procedure, others improve it by deriving new heuristics based on domain knowledge and problem analysis."
> "In general, operationalization converts knowledge about a task domain into procedures useful in performing the task. In this sense, AI researchers are engaged in operationalization when they convert domain knowledge into intelligent programs. Typically this involves taking a problem expressed in the language of a particular task domain, together with knowledge about the domain and reformulating them to fit a general computational method like heuristic search. This process can be viewed as mapping the problem into a call on a general procedure for the method, by finding suitable values for the arguments (which may include generators, tests, and search ordering) "- [Mostow,1983a]

Both Mostow's approach [Mostow,1983a,1983b] and our approach for advice learning focus on the issue of *change of representation.* The difference is that the change of representation in GE can happen on either the problem representation or the advice representation. Mostow's approach is mainly about the changes of the advice represen-

tation. Mostow's transformation approach is an automatic approach. Our work is an interactive approach.

6.3.2. Algorithm Design

Like Mostow's algorithm design method [Mostow,1988a], our editor can also perform algorithm design by the following steps:

1. build initial generate and test algorithm
2. improve the initial algorithm by transformation

The differences lie in the method of transformation. Mostow uses rule-based transformations. We use editing-based transformations. Rule-based transformations are suitable when domain knowledge is complete, but it's not as flexible as editing-based transformation. The following is a comparison between Mostow's rule-based transformation and our editing-based transformation for algorithm design.

- ☐ Mostow's approach requires complete domain knowledge. Ours can be used with incomplete domain knowledge, it can even be used without a priori domain knowledge.
- ☐ In Mostow's work, the transformations are mainly to speed up an algorithm. In our approach, the initial algorithm is allowed to be inefficient or even incorrect. The search for correct and efficient algorithms is achieved by revising the initial algorithm step by step.
- ☐ Mostow's approach requires all the domain knowledge to be given before problem solving. Ours allows the domain knowledge to be given incrementally during problem solving.

- Our editing approach is a kind of syntactic approach. When the intensions of primitive concepts are fixed, "change of representation" actually involves structural configuration. The editing approach is a powerful method in this case. When "change of representation" involves changes in the primitives, rule-based transformation is more suitable than editing approach.
- One problem which restricts the use of rule-based transformation is the difficulty of acquiring and formalizing rules necessary for problem solving. The editing approach has no such burden.
- Mostow's algorithm design is limited by the given transformation rules. It is suitable for design when the transformation rules are general enough to be applicable to all the design instances. This restricts the application of the transformation approach to a set of problems for which there exist a set of rules which are applicable to every problem. The editing approach is more flexible. It can do incremental design, it can also do creative design.
- Mostow's initial algorithm is composed of generators, tests, mappings and sorts. In our work, the initial algorithm can be composed of three kinds of components—generators, tests and functions. We treat all kinds of mapping as functions. The sorts mentioned in Mostow's work are a kind of ordering. In our work, for compound generators, some re-ordering can be accomplished by changing composition structures. For primitive generators, sorting can be done by adding a function on a collection of values.
- When the resulting algorithms prove to be incorrect, it's not easy to fix bugs in Mostow's approach. Mostow's idea of fixing bugs and deficient coverage by

localizing the responsible steps, repairing them and replaying them is similar to ours, but the method is not as simple and systematical as ours.

Mostow tried to solve this problem by relaxing constraints. But this method can help only when it's clearly known that the incorrectness of the resultant algorithm is exactly from being over-constrained and it's also clearly known which constraint to relax. Furthermore, the incorrectness of an algorithm may have various causes, over-constraint is only one of them.

In our editing approach, it's easy to remedy an incorrect algorithm. At any transformation step, the algorithm being transformed is allowed to be incorrect and inefficient. The algorithm incrementally evolves towards correctness and efficiency.

Mostow's	***Ours***
transformation rules are given before design	transformation rules are implicit in the agent's advice
all the constraints are given before design	constraints are given dynamically
control strategies are fixed	control strategies are changeable interactively
the seach goal is fixed	agent can dynamically shape or revise his search goal
automatic design	interactive design

Mostow has noticed the limitation of his approach, and in some of his latest work, he made his effort on semi-automatic approach by allowing user's interaction in rule-based transformation [Mostow,1988a]. Comparing our work with Mostow's semi-automatic design, the similarity is that we pay attention to human factors in the design process.

The relation is that the man-machine interaction is one kind of interaction supported by GE. The difference from the point of human interaction is that Mostow's method places a strong requirement on the user's knowledge. But our editing approach greatly eases the user's burden.

6.3.3. Algorithm Representation and Interpretation

On the topic of search algorithm representation, Mostow's previous work used a graphical notation. Mostow thought that the definition of graph notation "keeps changing, making it difficult to write a stable interpreter" [Mostow,1988b]. So, Mostow proposed a new representation called a "schema". Mostow pointed out that [Mostow,1988b]:

> The more important uses of the schema are
>
> - To enable the transformation rules to refer to algorithm components by name
> - To define the operational semantics of classes of algorithms by writing interpreters for them without having to cover the whole language

Unfortunately, a *schema* is not better than a graphical notation. Actually, a schema raises more problems than graphical notation does. The main uses of schema mentioned above by Mostow are not particular to the schema representation. The graphical notation has these two advantages as well. The difficulty of writing a stable interpreter for a graph is not because of the graph keeps changing. Instead, the trouble is from the kinds of *"primitives"* used to construct transformation rules. When we select the "primitives" only on the "invariant" factors of graphical notation, the design of stable interpreter becomes an easy task.

In our GE editor design, we use a tree structure which is the instance of graph structure. The *"invariant primitives"* we used for the algorithm transformation are the *composition structures* among components. In our editor, there are seven composition structures which can construct an infinite number of trees. What we need to do is to design an interpreter based on the seven types of composition structures.

Assuming that the schema of an algorithm is given, the sets of slots are fixed, and the schema is complete and correct. Our interpreting facilities bear a lot of similarities to Mostow's. In particular, the interpretation of composition structures are inspired by Mostow's interpretation descriptions [Mostow,1988b] although some details are slightly different.

Mostow's	*Ours*
Transformation works on declarative algorithm representation	transformation works on operational algorithm representation
schema is restricted to a fixed set of named components and must be extended to add any new ones	the set of components is not fixed. It can be easily changed
need a separate interpreter for each schema	need a separate interpreter for each composition structure. The algorithms are interpreted in a unified way.

6.4. Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) involves a set of n variables X1,...Xn, each represented by its domain values, R1,...Rn and a set of constraints. A constraint Ci(Xi1,...Xij) is a subset of the Cartesian product Ri1×...×Rij which specifies which values of the variables are compatible with each other. A solution is an assignment of

values to all the variables which satisfy all the constraints and the task is to find one or all solutions [Dechter,1989].

Interactive Search (IS) also involves a set of variables and a set of constraints. The differences between CSP and IS are the following:

1. In CSP, the constraints are given before search; the variable set is previously given and fixed during search; the variables are independent of each other before applying constraints; the variables are primitives without microstructures.
2. In IS, the constraints can be given during search; the variable set can be dynamically changed; the variables can have other dependencies beside the relationships imposed by constraints; the variables can be compound with complex microstructures.

6.5. Reinforcement Learning

Reinforcement Learning is a kind of learning through trial-and-error [Sutton,1990]. In the reinforcement learning paradigm, a learning agent receives from its environment a scalar performance feedback called reinforcement after each action. The agent's task is to construct a transfer function (called a policy) such that its performance is maximized [Lin, 1991].

Both Reinforcement Learning (RL) and Interactive Search (IS) have incremental and dynamic processing capacities. The main differences between RL and IS are the following:

1. The information received by RL system is only trial-and-error feedback (reward & punishment). The information received by IS system is agent's advice which can be the requests for viewing the system's dynamic states; the commands to change the system; the constraints or the judgements of the system's performance, etc.

2. In RL, the internal changes are only on the evaluation function and the policy. The set of state variables and the value range of each variable are previously determined and fixed during learning. In IS, the internal changes can be any aspects of the system's constructive structures; the control strategies and the search goals. The set of variables and the value range of each variable can be dynamically determined and changed.

Chapter 7. Conclusion

- *IHGT is efficient and flexible.* In GE, all the three search factors (search space, search control strategy and search goals) are dynamically changeable. Such deep level flexibility stems from the capacity of exploiting external knowledge, which results in the flexibilities at all the abstract levels in GE. Efficiency can be improved incrementally by dynamically acquiring external knowledge from agents and incorporating the knowledge acquired as changes of the search factors.
- *GE has broad applications.* GE is suitable for solving real world design problems including routine designs and non-routine designs. Design solutions provided by GE are hierarchical decompositions of design functions. The initial design problem solver for a particular domain is created by assembling components using a priori design constraints. The refinement of an initial design problem solver is achieved by taking agent's advice. The design goals are incrementally acquired and changed through the agent's advice.

  As a general-purpose search tool, *GE* is useful not only for the real world design tasks, but also for AI research. Because GE is a bottom level language shell of HGT, all the AI applications in HGT environment can be built at different abstract levels by taking GE commands as primitive operators. For example, natural language interface of HGT search can be built by transforming natural language sentences into GE commands step by step.
- *Syntactic (Editing) approach is very powerful*

  GE demonstrates that a syntactic approach to the issue of change of repre-

sentation is simple and powerful. GE makes HGT flexible without using any semantic information. Many complex learning tasks can be simplified as a sequence of operations on structural configuration if we carefully chose the primitive operators for reasoning. The editing approach for change of knowledge representation performs well in such cases.

References

[Braudaway,1989a]

Wesley Braudaway, Chris Tong, Automated Synthesis of Constrained Generators, Eleventh International Joint Conference on Artificial Intelligence, August,1989, Vol 1.

[Braudaway,1989b]

Wesley Braudaway, Constraint Incorporation and the Structure Mismatch Problem, in "Change of Representation and Inductive Bias", Paul Benjamin (Editor), Kluwer Publishing,1989

[Dechter,1989]

Rina Dechter and Judea Pearl, Tree Clustering for Constraint Networks, Artificial Intelligence 38 (1989) 353–366

[Flemming,1989]

Ulrich Flemming, Robert F. Coyne, Timothy Glavin,Hung Hsi , and Michael D. Rychener, A Generative Expert System for the Design of Building Layouts, Technical Report EDRC 48–15–89

[Holte,1986]

Robert C. Holte,R. Michael Wharton, Generative Structure in Enumerative Learning System , Proceedings of Sixth Canadian Conference on Artificial Intelligence, May, 1986.

[Holte,1988]

Robert C. Holte, An Analytical Framework for Learning System, Ph.D dissertation, Brunel University, England, Available as technical report AI88–72 from the Computer Science Department , University of Texas at Austin , Austin, Texas 78712.

[Holte,1989]

Robert C. Holte, Efficient Candidate Elimination Through Test Incorporation, in Change of Representation and Inductive Bias, Paul Benjamin (Editor), Kluwer Publishing,1989

[Lan,1990]

M. S. Lan and R. M. Panos, Printing Press Configuration: A Knowledge-based Approach, IEEE EXPERT, FEB. 1990, p65–73.

[Lin, 1991]

Long Ji Lin, Self-Improvement Based on Reinforcement Learning, Planning and Teaching, Proceedings of the Eighth International Workshop (ML91),323–327.

[Mitchell,1983]

Tom M. Mitchell, Paul E. Utgoff, Ranan Banerji, Learning By Experimentation: Acquiring and Refining Problem-Solving Heuristics, in R. S. Michalski, J. G. Carbonell, and T. M.. Mitchell (eds). Machine Learning , An Artificial Intelligence Approach, Tioga, Palo Alto, CA, pp163–1990,1983

[Mostow,1983a]

David Jack Mostow, Machine Transformation of Advice into a Heuristic Search Procedure, in Machine Learning: An Artificial Intelligence Approach, Michalski, R. C., Carbonell,J. G., and Mitchell, T. M. (eds), Tioga, Palo Alto, Calif., 1983.

[Mostow,1983b]

Jack Mostow, A Problem-Solver for Making Advice Operational, 3th National Conference on Artificial Intelligence, Washington, D. C.,Aug. 1983.

[Mostow,1988a]

Jack Mostow, A Preliminary Report on DIOGENES: Progress towards Semi-automatic Design of Specialized Heuristic Search Algorithms, ML-TR-27, Rutgers AI/Design Project working paper Number 108–1

[Mostow,1988b]

Jack Mostow, An Object- Oriented Representation for Search Algorithms, Rutgers AI/Design Project Working Paper number 107,1988.

[Ochs,1983]

E. Ochs Keenan and Bambi B. Schiefelin, Topic as a Discourse Notion: a Study of Topic in the Conversations of Children and Adults, in Acquiring Conversational Competence, Routledge&Kegan Paul, 1983, p90–p104 (Section 5.5.4. On Identifying Referents in the Discourse Topic).

[Rolston,1988]

David W. Rolston, Principles of Artificial Intelligence and Expert Systems Development, McGraw-Hill Book Company,1988,p68–69

[Sutton,1990]

Richard S. Sutton, Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming, Proceedings of the Seventh International Conference on Machine Learning, 216–224.

[Wile,1982]

D. S. Wile, Generator Expressions, Technical Report , USC Information Science Institute , Jan. 1982, Unpublished.

Appendix A: GE Commands

The specification of each type of command are given below in the form of tables. There are two columns in each table. The left column is for the specification of the command syntax. The right column is for the specification of the corresponding command function.

In this section, in each command formula, we use "+" to represent input arguments, use "-" to represent output arguments, use "?" to represent the arguments which can be either input or output. "+","-","?" are only used for the description, they are not the part of the commands.

G,Gi (i=1,2,...) represent generator names.

T,Ti (i=1,2,...) represent test names.

F,Fi (i=1,2,...) represent function names.

Glist represents a list of generator names.

The main purpose of this section is to provide a language specification of GE commands. The contents are mainly about the syntax and the functionality of each command without the details of each command translation. Some command interpretation algorithms will be given briefly as examples of how advice language is translated into a sequence of bottom level changes of HGT search.

Terminology: *Unattached Components* (also called *Free Components*) are the components which are not used in any generator hierarchies. Unattached components can be removed without any side effects. *Attached Components* are the components which are used in some generator hierarchies.

A.1. Viewing Dynamic Data

The "look" command is designed for viewing all the dynamic data, including generator records, function definitions, test definitions and global constants. It serves as the facility for agents to view the current context of the search system.

look(all)	Viewing all generator records in generator space
look(gen,+G)	Viewing the generator record of G
look(func,+F)	Viewing the function definition of F
look(test,+T)	Viewing the test definition of T
look(trigger,+G)	Viewing all the trigger tests attached to the generator G
look(constant,+Name,-Value)	Viewing the value of a constant by name
look(gfn,+G,-GFN)	Viewing the generative function GFN of the generator G
look(struct,+G,-Str)	Viewing the composition structure Str of the generator G. When G is primitive, Str returns nil.
look(para,+G,-P)	Viewing the parameter P of the generator G
look(comp,+G,-C)	Viewing the children C of the generator G
look(parent,+G,-Pt)	Viewing the parent generator Pt of the generator G
look(state,+G,-S)	Viewing the current generator state S of the generator G
look(result,+G,-R)	Viewing the current output result R of the generator G
look(outputname,+G,-Names)	Viewing the name assignments Names of output value of the generator G
look(parabond,+G,-PB)	Viewing the parameter dependency definition PB of the generator G.

The command "look(parent,G,Pt)" and "look(comp,G,C)" can be used to move up and down in the generator trees.

A.2. Checking

The "check" command is for checking some attributes of dynamic data. For all the "check" commands, the commands succeed when the checks are successful. The commands fail when the checks fail.

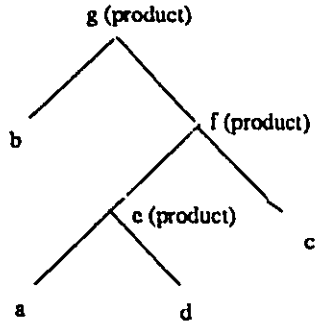
check(gen,+G)	check whether the generator G is defined or not.
check(func,+F)	check whether the function F is defined or not.
check(test,+T)	check whether the test T is defined or not.
check(root,+G)	check whether the generator G is a root generator or not. A root generator is a generator which has no parent.
check(leaf,+G)	check whether the generator G is a leaf generator or not. A leaf generator is one with no children.
check(freefunc,+F)	check whether the function F is free or not.
check(freetest,+T)	check whether the test T is free or not.
check(consistent,+X,+Y)	check whether term X and Y are structurally identical. It can be used to check the consistent connection among components. [a,2,[x,y]] and [b,3,[m,n]] are consistent but [a,b] and [a,b,3], "2" and "a" are not.

Unlike the "look" command which just views the dynamic data represented explicitly, the "check" command makes use of the dynamic data to do simple reasoning.

A.3. Finding

find(root,+G,-Root)	search for the root node in the generator hierarchy containing generator G.
find(root,-Roots)	return all the root nodes in the generator space.
find_loc_by_value(+Term,+Subterm,-Loc)	find the location of Subterm in Term. It returns the location of the first occurrence if Subterm occurs in Term more than once.
find_gen_by_loc(+Root,+Loc,-G)	locate the generator generating the entity corresponding to the location of an entity in the output values of generator Root.

find_gen_by_value(+Root,+Value,-G)	locate the minimal component G from which the entity "Value" in the output of Root is generated. When Value occurs more than once in the output, the command locates the generator G generating the first occurence.
find_gen_by_name(+Root,+Name,-G)	find the target component G by an entity name Name in the output result of the generator named Root.
find(mg,+G1,+G2,-MG)	find the minimal generalization MG of the generator G1 and G2.



Suppose the current value of g is [j,[[2,p],7]], j is from b, 2 is from a, p is from d, 7 is from c. We have:

Value	*Loc*	*Target*
j	[1]	b
2	[2,1,1]	a
p	[2,1,2]	d
7	[2,2]	c
[2,p]	[2,1]	e
[[2,p],7]	[2]	f
[j,[[2,p],7]]	[]	g

```
| ?- find_loc_by_value([j,[[2,p],7]],[2,p],L).
L = [2,1]
| ?- find_gen_by_loc(g,[2,1],G).
G = e
```

```
| ?- find_gen_by_value(g,[2,p],G).
G = e
```

If we give the names [n1,[n2,n3]] to the output value of g, we have:

```
| ?- set(name,g,[n1,[n2,n3]]).
yes
| ?- find_gen_by_name(g,n1,G).
G = b
| ?- find_gen_by_name(g,n2,G).
G = e
| ?- find_gen_by_name(g,n3,G).
G = c
```

The method for finding the root is to find the parent recursively until reaching one without a parent.

"find_gen_by_loc(+Root,+Loc,-G)" locates the generator G by its location Loc in the output value of Root. The method of doing this is a syntactic approach by structural decomposition.

"find_gen_by_value" command will stop searching when it reaches the "infunc", "outfunc" and "cascade" structures which are non-decomposable by locating output value syntactically. In this case, the target found might not be minimal. The minimal target can always be found by command "find_gen_by_name".

A.4. Statistics

time(+Call,-Time)

This command returns the execution time of a call which can be used to evaluate the time performance of a plan for statistics or for decision making.

A.5. Building a Generator from a Generative Function (GFN)

The "build" command is for creating generator instances from specific GFNs.

build(+Mode,+Arg1,+Arg2,?Arg3)	
build(gen,+GFN,+P,?G)	create an instance generator G from a GFN with parameter P. The actual result of this command is to build the corresponding generator record in the "initial" phase.
build(gens,+GFN,+N,?Glist)	create N generators in Glist with the same GFNs

The "build(gen,+GFN,+P,?G)" command creates the following generator record:

For a primitive GFN:

```
generator(G,GFN,P,[],nil,(initial,_,_),[_,[]]).
```

For a structural GFN:

```
generator(G,GFN,_,P,nil,(initial,_,_),[_,[]]).
```

The differences between these two generator records are in the parameter and component entities. In a primitive GFN, P is the parameter of the GFN and the component list is empty. A structural GFN is the GFN for creating composition structures. In a structural GFN, P is the component list and the parameter of the GFN is undefined initially.

A.6. Setting

The "set" command is for setting some global data.

set(constant,+Name,+Value)	set the value of a constant to a constant name.
set(name,+G,+Namelist)	set the name assignment of the output value of G to Namelist.
set(parabond,+G,+PB)	set the parameter dependency definition of the generator G to PB.
set(switch,+G)	set a switch to stop the value stepping on generator G if the current value is not consumed.

A.7. Modifying Dynamic Features of Generators

All the information stored in the generator records can be changed by "modify" command. The "modify" command can also be used to change the surrounding states when a generator's state is changed.

modify(+Mode,+Arg1,?Arg2)	
modify(name,+G1,?G)	change the generator name "G1" to "G" in generator record.
modify(struct,+G1,+Str)	change the old composition structure of G1 to new structure Str and initialize G1.
modify(para,+G1,+P)	change the parameter of G1 to P, and initialize G1.
modify(comp,+G,+Comp)	change the components of the generator G to Comp.
modify(parent,+G,+Pt)	change the parent of the generator G to Pt.
modify(state,+G1,+S)	change the current state of the generator G1 to the new state S.
modify(phase,+G,+PH)	change the transition state of the generator G to the value PH
modify(result,+G,+R)	change the current generated value of the generator G to value R.
modify(outputname,+G,+N)	change the name assignment of the generator G's output value to N.
modify(parabond,+G,+PB)	change the parameter bonding of the generator G to PB. PB=[G(E),G1(E1),...,Gn(En)], Gi (i=1,...,n) are the components. E, Ei (i=1,...n) are the parameter expressions.
modify(s_state,+G)	change the surrounding states of the generator G including parent state change and brother state change.

As bottom level commands, the modification is localized to the given generator without considering the surrounding components in the generator hierarchies.

For example, for the name change, if the old name is an attached generator, that is the component of another generator, the attachment between the old name and its ancestor generator remains unchanged. After name changing, the corresponding attachment leads to the old name which refers to an empty component. For example, suppose g1 has components [g2,g3], g2 has the components [g4,g5]. When the generator g2 changes the name to g6, because the change is only in the generator record of g2, there are no changes in the generator records of g1, g4 and g5. Consequently the children of g1 and the parent of g4, g5 still point to g2. Because every name in a component list must refer to an existing component, the agent needs to consider the corresponding change for consistency after name changing with this command.

A.8. Removing Components

The "remove" command removes the dynamic definition of components (generator, function, test) and some global data.

remove(all) ***remove(+Mode,+Arg1)*** ***remove(+Mode,+Arg1,+Arg2)***	
remove(all)	remove all the generator records in the dynamic space.
remove(gen,+G)	remove the generator G without considering the surrounding consistent change.
remove(gen0,+G)	remove the record of the generator G from the generator space with checking of surrounding conssitency and asking agent's confirmation.
remove(tree,+G)	remove all the generator records involved in the subtree rooted by G.
remove(func,+F)	remove the definition of a function F
remove(test,+T)	remove the definition of a test T.

remove(trigger,+G)	remove all the trigger tests attached to the generator G.
remove(trigger,+G,+Expr)	remove the trigger test corresponding to the advice expression Expr which is attached to generator G.
remove(switch,+G)	remove the switch to open the value stepping.

As a bottom level editing operation, removal takes place without considering consistency (except the command "remove(gen0,G)" which asks the confirmation from agents).

To make a consistent change or to do complex editing, checking (see "check" command) after or before removal might be useful. For example, to check whether a generator is:

1. compound or primitive

2. attached or unattached

When the given generator is primitive and unattached, the removal has no side effect. When G is attached and the record of G is removed, the attachment remains. Suppose G1 is the component of G which is in "product" structure, after G1 is removed, the name of G1 is still in the component list of G.

Similarly, the removal of a function works only on the function definition without considering whether the function is used by any generator hierarchies or not.

A.9. Building a Generator by Composing Components

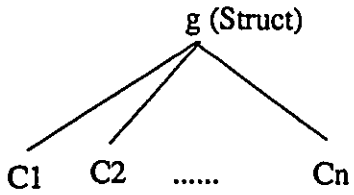
The "compose"command is for creating compound generators by composing components in a specified composition structure.

compose(+Struct,+Arg1,?Arg2)	
compose(infunc,[+F,+G1],?G)	create a new generator G by composing an input mapping function F and a generator G1.
compose(outfunc,[+G1,+F],?G)	create a new generator G by composing a generator G1 and an output mapping function F.
compose(test,[+G1,+T],?G)	create a new generator G by composing a generator G1 and a test T.
compose(product,+Glist,?G)	create a new generator G by composing a list of generators in Glist in a "product" structure.
compose(cascade,+Glist,?G)	create a new generator G by composing generators in Glist in a "cascade" structure .
compose(concatenation,+Glist,?G)	create a new generator G by composing the generators in Glist in a "concatenation" structure.
compose(concurrent,+Glist,?G)	create a new generator G by composing the generators in Glist in a "concurrent" structure.

The "compose" command composes the components in Arg1 with structure Struct to form a new generator named by Arg2. That is to create the following generator record:

generator(Arg2,Struct,_,Arg1,nil,(initial,_,_),[_,[]]).

Suppose Arg1 = [C1,C2,...,Cn], Arg2 = g. The command forms the following hierarchy:



Parameter binding can be defined for "product", "concatenation" and "concurrent" structures by giving parameter expressions after generator names in the following form:

compose(Mode,[G1(E1),G2(E2),...,Gn(En)],G(E))

Ei (i=1,...,n) and E are parameter expressions. Gi (i=1,...n) and G are generator names. Mode can be "product", "concatenation", and "concurrent". For example:

compose(product,[g1(X*[Y-2]),g2(Y)],g3([X,Y])).

A.10. Building a Generator by Making a Copy

copy(+G1,?G2)

This command makes a copy of G1 in G2. That is, it copies the generator hierarchy rooted at G1. The copying is done by recursively copying G1's sub-trees and its own generator record.

A.11. Generating Values by Firing a Generator

next(+G,-PH,-GR)	return the next value of the generator G in GR. When G is exhausted, PH returns "exhausted". Otherwise, PH returns "normal".
next0(+G,-PH,-GR)	the same as the "next" command except that the stepping will cause the surrounding state change.
next_n(+G,+N,-Vs)	returns the next N values of the generator G in Vs.
next_all(+G,-Vs)	collect all the rest of the values from the current state of the generator G into Vs.
look_ahead_one(+G,-PH,-V)	look ahead one value V and corresponding transition phase PH without state change.
look_ahead_all(+G,-Vs)	look ahead all the rest of the values Vs of G without state change.

There are four modes for generating the values of a generator :

- collecting mode
- stepping mode
- look-ahead-one mode

- look-ahead-all mode

Collecting mode outputs a sequence of values ("next_n"and "next_all" commands). *Stepping* mode outputs one value each time when a generator is fired ("next" and "next0" commands). *Look-ahead-one* mode outputs the next value without state change ("look_ahead_one" command). *Look-ahead-all* mode outputs all the rest of the values without state change ("look_ahead_all" command).

In most cases, the generating sequences are too big to explore completely. In such cases, the pruning of the search space becomes very important. Stepping mode is very useful for the dynamic pruning of search space.

There are some generator operations which work on a sequence of values instead of a single value. For example, sorting. One-time mode and look-ahead-all mode are useful for collecting values for the control of further search when the generating sequences of the generators are small.

Among all the value output commands, "next" is the essential one, all other commands are built from it. For example, "look_ahead_one(+G,-PH,-GR)" is implemented as follow:

1. copy G to G0
2. next(G0,PH,GR)
3. remove G0

A.12. Resetting a Generator

reset(+G)	initialize the generator G in order to generate values beginning from the first value.
reset0(+G)	the same as "reset" except that after resetting, the first value will be generated and the surrounding states will be changed.

A.13. Attaching Components into a Generator Hierarchy Using a Specified Structure

attach(+Struct,+Arg1,+Arg2)	
attach(infunc,+G1,+F,?G)	attach the input mapping F to the generator G1.
attach(outfunc,+G1,+F,?G)	attach the output mapping F to the generator G1
attach(test,+G1,+T,?G)	attach the test T to the generator G1.
attach(product,+G1,+G2,?G)	attach a new generator G2 to G1 in a "product" structure.
attach(cascade,+G1,+G2,?G)	attach a new generator G2 to G1 in a "cascade" structure
attach(concatenation,+G1,+G2,?G)	attach a new generator G2 to G1 in a "concatenation" structure.
attach(concurrent,+G1,+G2,?G)	attach a new generator G2 to G1 in a "concurrent" structure.

Unlike the "compose" command which uses components to build new hierarchies, "attach" command adds components into an existing hierarchy.

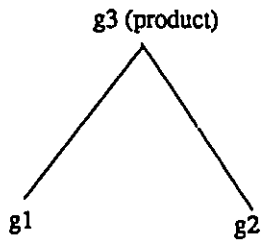


Fig. 1

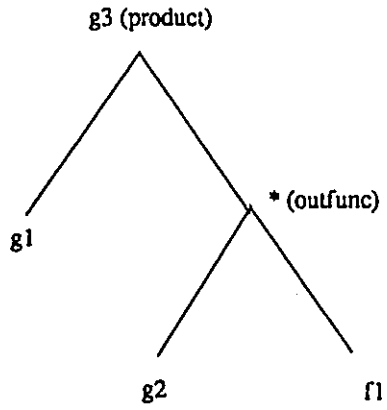


Fig. 2

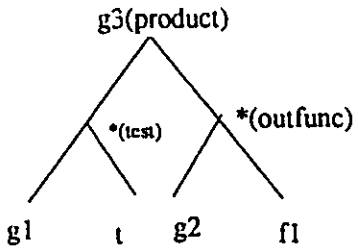


Fig. 3

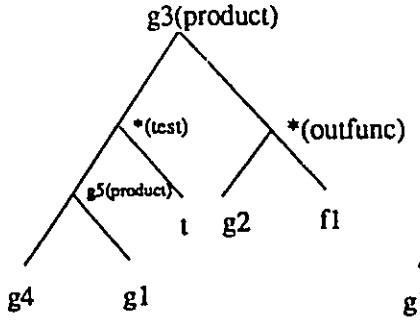


Fig. 4

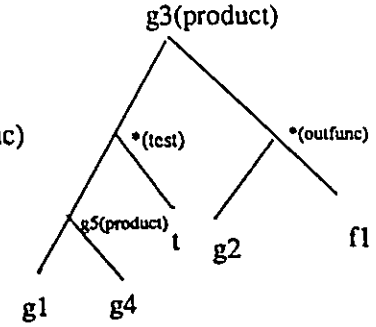


Fig. 5

"*" represents the names created by the editor when the names are not specified in the command.

By attach(outfunc,g2,f1,G), the tree in Fig. 1 becomes Fig.2.

By attach(test,g1,t,G), Fig.2 becomes Fig.3.

By attach(product,g4,g1,g5), Fig.3 becomes Fig.4.

By attach(product,g1,g4,g5), Fig.3 becomes Fig.5.

A.14. Detaching Components from Generator Hierarchy

detach(+Mode,+Arg1) ***detach(+Mode,+Arg1,+Arg2)***	
detach(gen,+G)	detach the generator G in the generator hierarchy from its parent.
detach(infunc,+G)	detach the input mapping function directly ahead of G.
detach(infunc,+G,+F)	detach the function F from the sequence of input mappings of G.
detach(outfunc,+G)	detach the function that directly follows G.
detach(outfunc,+G,+F)	detach the function F from the sequence of functions following G.

detach(test,+G)	detach the test that directly follows G.
detach(test,+G,+T)	detach a test named T from the test sequence following G.

The "detach" command is to detach a component from a generator without any change in the detached component. After detaching, the detached component will be a free component. "detach" is the inverse of "attach".

By detach(gen,g4), Fig.5 becomes Fig.3.

By detach(test,g1), Fig.3 becomes Fig.2.

By detach(outfunc,g2), Fig.2 becomes Fig.1.

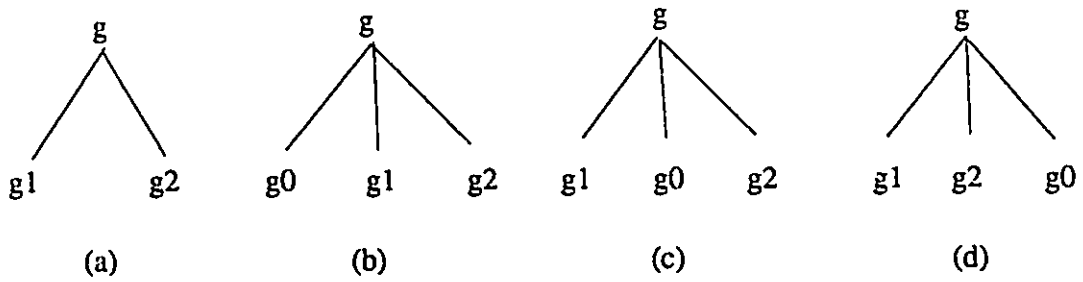
A.15. Inserting a Component into the Component List of a Generator

insert(+N,+G,+G1)

This command inserts generator G1 in the component list of G at the position N. Like the "attach" command, the "insert" command is for adding component into an existing generator hierarchy. The difference is that "attach" creates a new sub-hierarchy by taking G1 and G as components, but "insert" adds one new component into the component list of a generator.

The steps of translating this command are the following:

- ☐ insert G1 to the Nth position in the component list of G
- ☐ $parent(G1) \leftarrow G$
- ☐ create the new state of G based on the current components of G
- ☐ change the surrounding state of G



by insert(1,g,g0), (a) becomes (b).

by insert(2,g,g0), (a) becomes (c).

by insert(3,g,g0), (a) becomes (d).

A.16. Extracting a Component from the Component List of a Generator

extract(?N,+G,?G1)	
extract(+N,+G,-G1)	extract the Nth component in the component list of the generator G and return the extracted generator name in G1.
extract(-N,+G,+G1)	extract the generator G1 in the component list of the generator G and return the initial position of G1 in the component list in N.

G and G1 are generator names. N is an integer which represents a component position in the component list of G. This command extracts a component in the component list of the generator G. The main steps of translating this command are the following:

- ☐ take out G1 from the component list of G
- ☐ $parent(G1) \leftarrow nil$
- ☐ modify the current state of G based on the current components of G
- ☐ change surrounding state of G

Unlike the "detach" command, the "extract" command takes out a generator from component list using position information. "extract" is the inverse of "insert".

By extract(2,g,G), figure (c) becomes (a), and G returns g0.

By extract(N,g,g0), figure (d) becomes (a), and N returns 3.

A.17. Replacing Components

replace(+Mode,+Arg1,+Arg2) ***replace(+Mode,+Arg1,+Arg2,+Arg3)***	
replace(gen,+G1,+G2)	replace the generator G1 in the generator hierarchy by G2.
replace(infunc,+G,+F)	replace the input mapping function directly ahead of G by F.
replace(infunc,+G,+F1,+F2)	replace the function F1 in the input mapping function sequence of G by F2.
replace(outfunc,+G,+F)	replace the output mapping function directly following G by F.
replace(outfunc,+G,+F1,+F2)	replace the function F1 in the output mapping function sequence of G by F2.
replace(test,+G,+T)	replace the test directly following G by T.
replace(test,G,+T1,+T2)	replace the test T1 in the test sequence of G by T2.

The main steps of replacing C by C0

- ☐ replace C in the component list of parent(C) by C0
- ☐ if C0 is a generator,then
  $parent(C0) \leftarrow parent(C)$

  $parent(C) \leftarrow nil$
- ☐ create the state of parent(C0) based on the current components of parent(C0)
- ☐ change the surrounding state of parent(C0)

The command replace(gen,g2,g4) will make Fig.6 becomes Fig.7.

In Fig.8, the function sequence of g2 is [f2,f1]. The command replace(g2,f2,f3) replaces the function f2 in Fig.8 by f3.

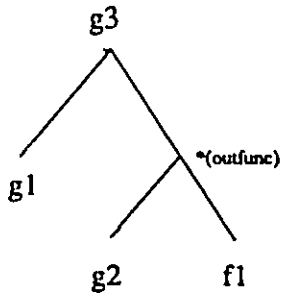


Fig. 6

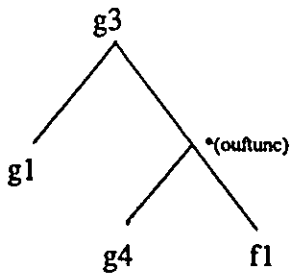


Fig. 7

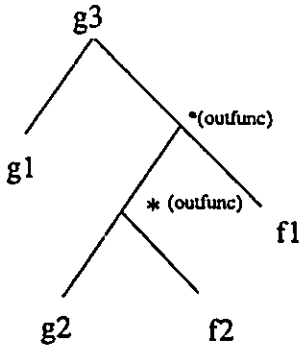


Fig. 8

A.18. Taking Advice

Advice(+G,+Expr)

G is the name of a root generator. Expr is the advice expression.

This command analyzes the advice expression Expr and set the corresponding trigger test to the root generator G.

Advice expression is in the following formal grammar:

Expr —> E OP E

OP —> >|>=|<|<=|=|\= .....

E —> T+E|T-E|T

T —> I*T|I/T|I

I —> name |number ....

name —> $string

Suppose g1 is an integer generator, its value name is "width", and we want to set the following constraint:

$width + 3 < 10

then the command is:

advice(g1,[$,width,+,3,<,10]).