BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-90-M11

An Implementation of Cooperative Concurrency Control in an OODB

by
Chen Hui-ching Liu

An Implementation of Cooperative Concurrency Control in an OODB

by
Chen Hui-ching Liu

Research Project

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science in the
Department of Computer Science at Brown University.

June 1990

This project by Chen Hui-ching Liu
is accepted in its present form by the Department of
Computer Science in partial fulfillment of the
requirements for the degree of Master of Science.

Date 7/11/90

Stanley B. Zdonik

Stanley B. Zdonik, Advisor

Acknowledgements

I would like to thank my advisor, Stanley B. Zdonik, and Andrea H. Skarra for their help in completing this project. Also, thanks to my husband, Jen-kuei, my parents, and my parents-in-law, without their support and encouragement I would not have made it through this time.

An Implementation of Cooperative Concurrency Control in an OODB

Chen Hui-ching Liu

June 1990

1 Introduction

This is an implementation for a model of transactions and concurrency control that supports cooperative data sharing among users. The context for this model is an object-oriented database system which includes an extensible data abstraction facility. This new model has some particular features which are different from the traditional model as follows.

1. The new model has a semantic concurrency control which supports a programmer-defined correctness criteria that uniformly integrate both data and application semantics instead of the traditional correctness criteria that contain a global consistency for every single transaction and the serializabilty for the concurrent transactions.

2. A group of cooperating transactions[1] (CTs) in this model is the unit of consistency instead of an individual transaction in the traditional model. A CT generates an operation invocation sequence that is dynamically defined, rather than statically defined.

3. A nested framework of transaction groups[2] encapsulate and localize nonserializable data sharing. The new model supports nested transac-

[1] A cooperating transaction is an initiator of operation invocations. It is a member of a transaction group and it is defined by an application programmer.

[2] A transaction group (TG) is a user-defined collection of CTs or other TGs that includes an explicit and semantic specification of the correctness criterion for the group's histories.

tion groups which transaction groups could contain other transaction groups. It extends the transaction theory to be multilevel.

4. The new model prevents the interaction between unrelated transactions and it permits only the prespecified interactions among cooperating transactions in order to enable users to reason about the correctness and integrity of each unit of work. The traditional transactions are isolated and unrelated interactions between users and a database that maintain global consistency. That is, each transaction in the traditional database system is atomic and concurrent transactions are serializable with each other. But in the new model, the users must interact not only with the database but also with each other.

5. The data objects manipulated by the traditional model are very large, nested, and interrelated by many consistency constraints. In general, the tasks are very complex, but the new model divides the task into simpler and parallel subtasks which can reduce the complexity.

The project is implemented for these purposes by the C programming language and is divided into 'DRIVER', 'CC', 'QUEUE', and 'PAT' four C modules. To begin the program, enter the main() function to select one option by the user in the DRIVER module. According to the different options, you need to give the different inputs by following the directions. Then the DRIVER module will call one of the interface functions (for more detail see section 3) with the CC module to enter the CC module. If the current option is to invoke an operation from the queue list, or to invoke an operation which is queued later, or to print current contents in the queue list, the CC module will call the interface functions (for more detail see section 4) with the QUEUE module to do the queue thing in the QUEUE module and then come back to the CC module. To enter each pattern machines, the CC module need to call the interface functions (for more detail see section 5) with the PAT module to enter the PAT module and then come back to the CC module. After the CC module finishes all the jobs for the particular option, it will go back to the DRIVER module to wait for the next input option from the user.

I illustrate the overall synchronization scheme here by tracing the path through the program of an operation invocation 'opinv' of a CT member or an object, the return of an operation result, and the commit of a TG. First,

let's talk about an operation invocation which may be put into the queue list or be sent to an object for execution. Once an operation is invoked by a CT member, the user needs to input the operation name, member path name, object name, and other arguments from the driver. The concurrency control module assigns a unique opinv identifier to the operation invocation. It opens a template entry for this opinv, it finds the patterns that are indexed by the member name and the object name in the FINDER data structure, and it sends the operation request into these patterns. If the operation request matches an arc in any of the patterns from the current state to a dead state, the information of the operation invocation is added into the queue list and the request is suspended. Otherwise the operation request is saved as a new element entry in CCinv and is sent to the object for execution.

Second, an object can return an operation result or invoke a suboperation. This case in which the object returns an operation result is described below. To simulation object invoking a suboperation, the user inputs the suboperation's operation name, object name, and other arguments, and also inputs the name and opinv of the object invoking the operation to the driver. The concurrency control gives the suboperation a unique opinv, finds the proper index in CCinv and finds the CT member name that invoked this suboperation. It opens a new template entry for this suboperation, and it finds the patterns (indexed by the member name and the object name in the FINDER data structure) which are not the same as any of the patterns matched by superoperations of this operation. It then sends the operation request to these patterns. If the operation request matches an arc in one of the patterns from the current state to a dead state, the information of the operation request is added into the queue list and the operation request is suspended. Otherwise, the concurrency control creates some subordinate patterns for this operation's superoperations according to different tg numbers, and the suboperation is sent to the object for execution.

Subordinate patterns are created during suboperation invocations and will be destroyed if they become inactive when the operation results are returned to their invokers. A subordinate pattern is a finite automaton whose alphabet consists of operation invocations on a single object and a TG's members. The subordinate pattern machine implements a conflict criterion, and it is an instance of a subordiante template that is defined by the object's type; each pattern machine's subordinates specify its invalidation at objects accessed by the suboperations.

Third, an operation result is returned to its invoker which could be a CT member or another operation's object name. The user just needs to input the unique opinv of the operation to be returned, and the type and the value of the result from the driver. The concurrency control sends the subordinate pattern names and the result to the matched patterns in the p_list of the opinv's CCinv entry. The pattern machines then return OK or ERROR to the caller in the concurrency control. If any of the patterns respond error, notify the object of opinv's CCinv entry to abort the opinv and exit. For each inactive subordinate pattern[3], 'sub', destroys it. Finally, send the operation result to the invoking object or CT, and clear the CCinv entry for the opinv.

Last, TG commit request adjusts the TG hierarchy if necessary. The user needs to Input the TG number to be requested to commit from the driver. The concurrency control will check if all the TG members are committed and the current states of all the patterns in this TG are in final states. If yes, the TG commit request is successful and the TG will be erased and no longer exist. Otherwise, the commit request fails and does nothing.

Figure 1 shows the relations between these C modules. I will describe their data structures and pseudocodes of localized functions in section 2 to section 5. And I include all the C programming codes and several examples at the end of the project.

[3]An inactive subordinate pattern is a subordinate pattern which has the local variable locvar[0] = 0. That is, count = 0. (See also section 5.)

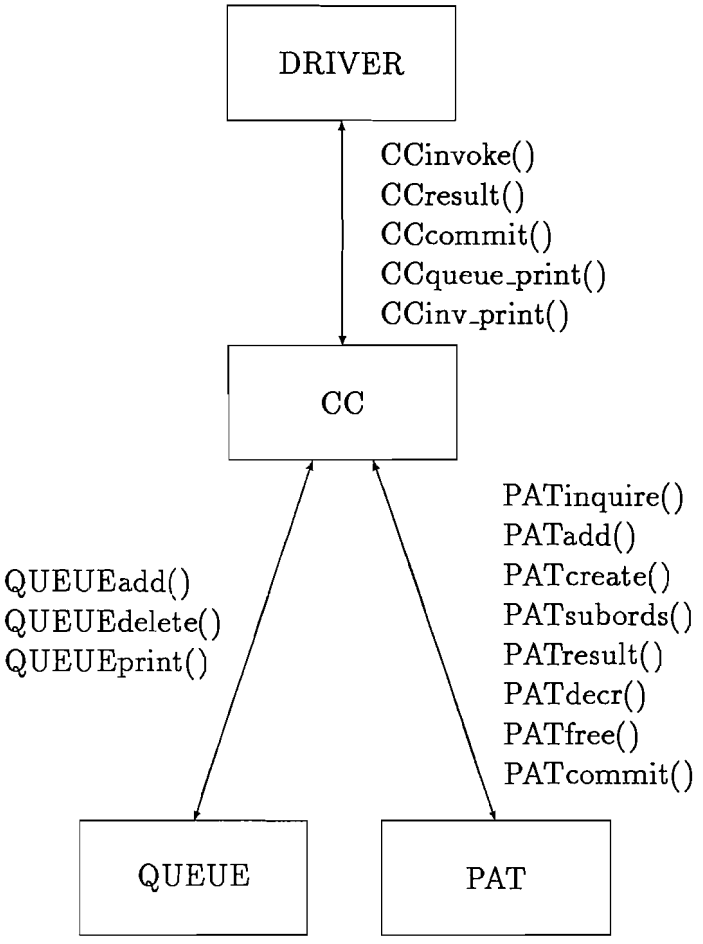


Figure 1: The interface of the C modules inside the project.

Section 2 describes the driver (the DRIVER module) which is the entry point of the program. The section points out the interface functions between the driver and the concurrency control part (the CC module).

Section 3 is the concurrency control which is the central module of the project and it interfaces with all the modules in the program including the driver, the queue list (the QUEUE module), and the pattern machines (the

PAT module). It also describes the specification of the user-defined correctness criteria for every TG and the TG hierarchy which is defined in data structure "CCfinder".

Section 4 is the queue list which is a special part of the CC module. The only module that interfaces with the QUEUE module is the CC module.

Section 5 is the pattern machines which contains the interface between the CC module and the PAT module, and is the only module to catch the data from the pattern machines which are the other user-defined part[4].

Section 6 contains the C programming codes which include all the C programs and their related "include" files except two user-defined parts.

Section 7 is the example part which contains two sample examples. Each example contains the TG hierarchy for all the TGs, the user-defined correctness criteria for each TG, the user-defined pattern machines, and some of their demo outputs.

2 The Driver

The driver is the simulation part for the "database", the "user", and the "queue". It simulates the database to invoke suboperations[5] and to return results; it simulates the user to invoke operations and to receive results; and it simulates the queue to invoke operations and to receive the operations for storage. The driver takes the inputs from the screen and calls functions in the CC module. In the driver, there is one data structure -- result, and it contains the "main" function of the project and two database simulation functions, GETinput() and GETresult() as follows.

2.1 The Data Structure in the DRIVER Module:

result : The data structure is for storing the operation result that is returned from the database to the invoker. The data structure is defined

[4]There are two user-defined parts in the project. They are the TG hierarchy which is included by the CC module and the pattern machines which are included by the PAT module.

[5]A suboperation is an operation which is invoked by another operation called 'superoperation'.

globally in the driver, and contains 'r_type' and 'uval' two fields as describe below.

r_type - an interger [0-4] to represent the types of the operation result.

$$r\_type = \begin{cases} 0 & \text{if the result is ERROR,} \\ 1 & \text{if the result is an integer,} \\ 2 & \text{if the result is a float number,} \\ 3 & \text{if the result is a long number, or} \\ 4 & \text{if the result is a char string.} \end{cases}$$

uval - a field to store the value of the operation result. The result could be an integer, a float number, a long number, or a char string.

2.2 main():

It contains a loop of interactions where each interaction has eight independent choices by inputing an integer option number [1 - 8]. The forms of an interaction are the following:

1. An operation is invoked directly by a CT member -- call function GETinput(option, opname, invoker, args) to get the operation invocation information and call function CCinvoke(option, opname, args, invoker) in the CC module to do the invocation.

2. An operation is invoked by the waiting operation list -- call function GETinput(option, opname, invoker, args) to get the operation invocation information and call function CCinvoke(option, opname, args, invoker) in the CC module to do the invocation.

3. An operation is invoked by the nearest superoperation's object -- call function GETinput(option, opname, invoker, args) to get the operation invocation information and call function CCinvoke(option, opname, args, invoker) in the CC module to do the invocation.

4. Return an operation result to the invoker -- call function GETresult(opinv) to get the returning result from the object and call function CCresult(result, opinv) in the CC module to return the operation result.

5. Commit a transaction group TG -- input the TG number, tg, to be commited and call function CCcommit(tg) in the CC module to do the commit request.

6. List all the operation information in the current queue list -- call function CCqueue_print() in the CC module to print the information in the queue list.

7. List all the important information that is currently in CC module's data structure "CCinv" -- call function CCinv_print() in the CC module to print the information.

8. Quit and leave the interaction loop.

2.3 GETinput(option, opname, invoker, args):

Inputs all the information that an operation invocation needs. That includes the operation name – opname, the invoker's pathname – invoker according to the different option number (from 1 to 3) given in the main function, and the argument values -- args where the object name is the first element in the args.

2.4 GETresult(opinv):

Inputs the operation invocation identifier 'opinv' of the operation to be returned and receives the operation returning result 'result' which could be an integer, a float number, a long number, a char string, or an ERROR result.

3 The Concurrency Control

The concurrency control, the CC module, is the correctness criteria manager. It globally defines the following four data structures -- CCfinder, CCinv, CCinv_last, and CCsubs. And it directly connects with the driver module, the pattern module, and the queue module. The driver calls five CC module's functions which are CCinvoke(), CCresult(), CCcommit(), CCqueue_print(), and CCinv_print(). In addition, the concurrency control calls some functions in the PAT module and in the QUEUE module, and I will describe the calling

conventions in more detail in the descriptions of the PAT module and the QUEUE module.

3.1 Data Structures in the CC Module:

They are globally defined in the CC module and exist while the concurrency control exists. In contrast, individual entries within the data structures are added or deleted as necessary.

CCfinder : The data structure keeps all the relationships in the current TG hierarchy among members, objects, and all the patterns[6]. The structure will exist all the time, and will be modified when a TG has been successfully committed or when a subordinate pattern is added or deleted. The data structure contains two indices into the patterns of the system and application. Each entry (see also in *Figure 2*) is defined for a unique TG and contains 'mem' and 'obj' two sub data structures as follows.

mem - an array of all members' records for one particular TG. Each record is for a member of the TG and contains the following two fields:

name -- the member name

ptns -- an array of records where each record contains the index of a matching pattern - "pat" and whether the pattern is extend[7] or not - "extend"

obj - an array of all objects' records for the same TG as above. Each record is for an object of the TG and contains the following two fields:

name -- the object name

ptns -- an array of the matching pattern indices

[6]The patterns include primary patterns and subordinate patterns. A primary pattern is a pattern which is given by the users and will exist all the time. A subordinate pattern is generated by the program to handle the conflict problem of the suboperations.

[7]A pattern is extended if the conflict between an operation and its suboperations is to be ignored.

(a) Data structure for each entry in CCfinder.

mem		obj	
name: string	ptns: array of part (b)	name: string	ptns: array of integer

(b) Data structure for each element of ptns in mem record.

pat: integer	extend: integer

Figure 2: Data structure for an entry of CCfinder.

CCinv : The data structure is for keeping the operation invocation information. If an operation is invoked by a CT member, a new element entry in CCinv will be opened. If the operation is a suboperation, the information will be retained in a new entry which has linked with its nearest superoperation. The data structure is an array of linked lists, where each array entry corresponds to an operation invocation from a CT member. A CCinv entry (see also in *Figure 3*) exists only during the execution of an operation and its suboperations, and the fields of each list entry are the following:

next_ptr - a pointer that points to its suboperation, if there is one, or a pointer that points to itself if it invokes some suboperation that has been queued, or null if there is no suboperation for the current operation entry

pre_ptr - a pointer that points to its nearest superoperation if there is one, or null if its invoker is a CT member

opname - the string name of the operation

args - an array of char strings where the object name is in the first element of args.

invoker - the invoker of the operation which is a CT membername or an object name

opinv - a unique operation invocation identifier

subords - a list of the subordinate patterns where each subordinate pattern has at least one superordinate pattern in the p_list

create_subs - a list of the subordinate patterns which are created when the operation is invoked

p_list - an array of pattern lists, where each pattern is indexed by matching the membername and the object in the CCfinder, and each pattern list contains the patterns according to different TGs. The fields of each pattern list are as follows.

tg -- the TG number

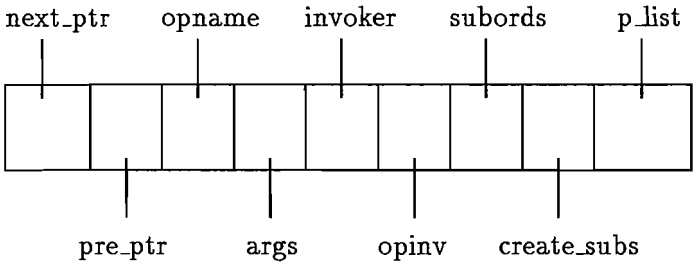
membername -- a string membername

duplicate -- an array of pattern indices which are found in one of the matched lists of the operation's superoperations

match -- an array of the patterns where each pattern contains the matched pattern's index and whether the pattern is an extend or not

nomatch -- an array of pattern indices where each pattern contains the nomatched pattern's index and whether the pattern is an extend or not

(a) Data structure for each entry of CCinv.



(b) Data structure for p_list in part (a).

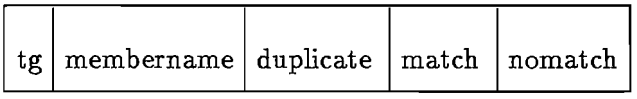


Figure 3: Data structure for CCinv.

CCinv_last : It has the same type of data structure as CCinv. The only difference between these two is that each element of the CCinv_last points to the last entry of the related element of the CCinv if the element of the CCinv is not null; or it is null otherwise.

CCsubs : The data structure is for storing superordinate patterns of all the subordinate patterns. The data structure is an array of entries where each entry corresponds to a subordinate pattern whose superordinate patterns are defined by the same TG and the same CT member. A CCsubs entry (see also in *Figure 4*) exists only if the subordinate pattern exists. The fields for each entry are the following:

subname - the unique pattern index of a subordinate pattern

tg - the TG number

extend - represents whether the subordinate pattern is an extend or not. If yes, extend = 1; otherwise, extend = 0.

opname - the string name of the operation which creates the subordinate pattern

object - the string name of the object

CT - the CT member name

opsub - an array of the superordinate pattern lists which separate the different operation into different opsub entry. The fields of each entry are as follows.

opinv -- a unique string name for the operation invocation identifier or a constant string "complete" to store the superordinate patterns of the operation whose result had been successfully returned.

superords -- a list of the superordinate patterns for the related opinv

(a) Data structure for an entry of CCsubs.

subname	tg	extend	opname	object	CT	opsub

(b) Data structure for opsub in part (a).

opinv	superords

Figure 4: Data structure for CCsubs.

3.2 CCinvoke(option, opname, args, invoker):

It is called if there is an operaton invocation from a CT member, from the queue list, or from another operation's object. For each operation invocation, we do the following:

- to generate a unique operation identifier based on the invoker,
- to find the CT membername that is invoked by the operation or its superoperation, and find the proper index 'i' in the CCinv for the new operation invocation,
- to enter the operation name, the arguments, the invoker, and the operation invocation identifier into a new entry −− template,
- to call the function 'FINDERreturn(CT, template, i)' (the FINDER part) to find all the possible patterns in CCfinder by matching both TG member and the object, and to put them in the p_list of the 'template',
- to send the operation request to each pattern in the nonduplicate list and to return CONFLICT, MATCH, or NOMATCH by calling the function PTNSreturn (template, CT, i);
- to save the template into the proper part of the CCinv if the function PTNSreturn() returns MATCH or NOMATCH; otherwise, call the function Suspend(template, opinv, i) to do the queue things and exit,
- if the operation invocation is a suboperation, call the function 'Create_Subords(i)' to create subordinate patterns for the patterns matched by the operation's superoperations. If not, the CCinv_last[i]'s fields 'create_subs[0] = -1' and 'subords[0] = -1'.

3.2.1 FINDERreturn(CT, template, i):

For every supergroup of the given CT, this function lists all the match patterns by using CT's related membername and the input object. It calls function FINDptns(template, tgindex, tgnum, i) to search and to list the patterns for one particular TG. It returns ERROR if the function FINDptns() returns ERROR; otherwise, it returns OK.

FINDptns(template, tgindex, tg, pindex): the description is as follows.

- Finds the member and the object location in the CCfinder for the particular TG number, 'tg'.
- Searches and lists the matched patterns by the membername and the object and put them in the template entry of CCinv[pindex], "template". If the matched pattern is duplicated with the superoperation's matched pattern, put the pattern in the duplicate list, 'template->p_list[tgindex].duplicate'. If not, put it in the match list, 'template->p_list[tgindex].match'.
- Returns ERROR if the membername does not exist in the CCfinder; otherwise, returns OK.

3.2.2 PTNSreturn(template, CT, pindex):

For each pattern returned by the FINDER, this function calls function PAT-inquire(pat, opname, args, membername) in the PAT module to return the result which is one of the MATCH, NOMATCH, and CONFLICT. If the result is MATCH, it is saved in the match list of 'template' if the pattern is not a subordinate pattern. If the result is NOMATCH, it is saved in the nomatch list in 'template' if the pattern is not a subordinate pattern. Otherwise, the result is CONFLICT, and if the pattern is a subordinate pattern and the operation is a suboperation, we call the function CONFLICTignore(P, pindex, template) to check whether the conflict can be ignored or not. If it can be ignored, the program will do nothing for this pattern and will go for getting the next matched pattern's result; otherwise, CONFLICT will be returned.

CONFLICTignore(P, pindex, template): checks if we can ignore the conflict for the operation. For most patterns, the sequence of operations they describe induces the ordering of the operation's suboperations. That is, the sequences described by these patterns extend to their suboperations. Thus, conflict that may occur between these suboperations can be ignored when the pattern does not define conflict among their respective superoperations.

- Check if the operation is a suboperation.
- Find the location in CCsubs of the subordinate pattern.

- Check if every superordinate pattern of this subordinate pattern is an extend.
- Check if every superordinate pattern of the subordinate pattern P[8] contains at least one of the superoperations of the operation. That is, for each superordinate pattern in CCsubs[P], find the pattern in CCinv[pindex[9]]->match or CCinv[pindex]->nomatch for some superoperation of the operation.
- If yes, return OK; otherwise, return FAIL.

3.2.3 Suspend(template, opinv, i):

This function calls the function QUEUEadd(opname, args, invoker, opinv) in the QUEUE module to send the operation request to the waiting operation scheduler and notify the invoker [CT or another operation's object identifier] that the current operation is queued. And then clear the template entry of CCinv, 'template'.

3.2.4 SAVEp(template, pindex):

This function appends the template entry of CCinv, "template", into CCinv[pindex].

3.2.5 Create_Subords(ccindex):

This function creates subordinate patterns for the primary patterns which are matched by the superoperations of the opinv. For each TG, tg, in the p_list field of opinv, if at least one of the matched patterns of the superoperations is non-duplicated with opinv, a new field is added in CCsubs if there exists one already or a new subordinate pattern is created otherwise. For each 'tg' in the operation we do the following:

- create a template space, "temp_sub", for an CCsubs entry
- for each matched pattern in the p_list of opinv's superoperations, if the pattern is not duplicated with the p_list.duplicate field of opinv, add the pattern into the proper field of temp_sub.

[8] P is the index of the subordinate pattern in CCsubs.
[9] 'pindex' is the index of the suboperation in CCinv.

- if we add one or more superordinate patterns into the temp_sub, then check if there already exists an entry in the CCsubs by using opname, object, membername, and tg. If no previous entry in CCsubs, a new entry in CCsubs will be created.
- if we create a new entry in CCsubs, we call function PATcreate (CCinv_last[index], pattern[subord], CCfinder[tg].mem, i, count) in the PAT module to create a new subordinate pattern, and add the new subordinate pattern into the proper place in the CCfinder.
- Otherwise, if we add a new field in CCsubs, for every superordinate pattern 'superpat' in temp_sub, we delete it from the temp_sub if it is already in the CCsubs' entry. If one or more superordinate patterns remain in the temp_sub, we call function PATadd(pattern[subord], count) to add the count of the superordinate patterns into the subordinate pattern.
- add the subordinate pattern into the 'subords' field of the operation's superoperations, and add the index of the CCsubs into the 'create_subs' field of CCinv_last[index]

3.3 CCresult(result, opinv):

CC module receives the result for invocation 'opinv' from the object.

- search the index in the CCinv structure
- if the result from the object is an error, then call function Abort(result, i, opinv) and exit
- send the subordinate pattern names and the result to the matched patterns in opinv's CCinv_last->p_list.
  - For each subordinate patterns in the CCinv->subords list of the opinv, read the CCsubs->opsub.superords.
  - To each matched pattern, P, in the opinv's CCinv_last->p_list, if the subordinate pattern's superordinate patterns contain P, call function PATsubords (P, CCinv_last->opname, object, CCinv_last->membername, subords) in the PAT module where the object's access generated the subordinates.

- Call function PATresult(P, opinv, args, result, inactive) in PAT module to send the result to each matched pattern P.

- send the result back to the invoking object or the CT member
- clear the CCinv entry for the opinv

3.3.1 Abort(result, i, opinv):

- assign the result to be ERROR
- notify each matched pattern 'P' of the operation that the returning result is ERROR by calling the function PATresult(P, opinv, result, waste) in the PAT module
- for each subordinate pattern 'sub' that the operation is created, call function Destroy(sub, 1, i) to destroy the subordinate pattern 'sub'
- send the operation request back to the invoking object or the CT member by calling the function RESULTback(result, opinv) and nodify the invoker that the returning result for the operation opinv failed
- clear the CCinv entry for the operation 'opinv'

3.3.2 Destroy(sub, option, index):

The option is an integer 0 if it successfully returns the result for the operation opinv, or integer 1 if it is called by the function Abort().

- if option = 1 : for the subordinate pattern in CCsubs[sub], we clear the opinv's entry and get the count number which is the superordinate pattern number for the operation, 'opinv'. And we call function PATdecr (CCsubs[sub]->subname, count) to decrement the local variable 'count' in the subordinate pattern. After the decrement, if the local variable 'count' is 0 then we change the option to be 0.
- if option = 0 :

  1. remove the subordinate pattern from the memory by calling the function PATfree(CCsubs[sub]->subname) in the PAT module

2. clear the references for the subordinate pattern in the CCfinder

- for each superoperation of the operation 'opinv', clear the subordinate pattern name from the subords field in CCinv
- if option = 0, clear the subordinate pattern's entry in the CCsubs

3.3.3 RESULTback(result, opinv):

This function returns the result back to the invoker where the result can be an integer, a float number, a long number, a character string, or an ERROR result.

3.4 CCcommit(tg):

It is a transaction group, tg, to request commit.

- If 'tg' has uncommitted subgroup[10], then return ERROR.
- For each CT member of tg, find all the patterns that are indexed in the CCfinder by this member. And for each pattern, P, call function PATcommit(P) in the PAT module to check if the current state of the pattern machine P is in final state.
- If any pattern machine returns nonfinal, then return ERROR.
- Remove the related index in the CCfinder and return OK.

3.5 CCqueue_print():

It is a bridge between the driver and the waiting operations. The driver calls this function to print the information in the current queue list and the CC module transforms the command and directly calls QUEUEprint(index) in the QUEUE module to print an element's information of the queue list at a time.

[10]An uncommitted subgroup is a tg member which is another transaction group.

3.6 CCinv_print():

It is a function to print the current important information of each entry in the data structure CCinv. For each entry we print the 'opname', 'args', 'invoker', 'opinv', and the matched pattern list in the p_list.

4 The Queue List

The queue list is also called a waiting operation or a heap. It can be regarded as a special part of the concurrency control and we store this part in the file "QUEUE.c". The CC module directly calls 'QUEUEadd', 'QUEUEdelete', and 'QUEUEprint' and the driver indirectly calls the 'QUEUEprint' via 'CC-queue_print'. The waiting operations contain one data structure, queue, and three functions, QUEUEadd(), QUEUEdelete(), and QUEUEprint(), as follows.

4.1 the Data Structure in QUEUE Module:

queue : The data structure is for storing all the information of operation invocations which have been queued. The data structure contains at most 100 queued operations. A queue entry (see also in *Figure 5*) exists when the operation is queued and will be erased when the queued operation is invoked again. The data structure is a linked list and each entry contains the following fields:

opname - the string name of the operation

args - a list of arguments for the operation where the first element in the args is the object of the operation

invoker - the string invoker of the operation. It could be a CT membername, or it could be the object identifier of the nearest superoperation of the operation.

opinv - a unique operation invocation identifier of the nearest superoperation if the operation is a suboperation, or string "0" if the invoker is a CT member

opname	args	invoker	opinv

Figure 5: Data structure for queue.

4.2 QUEUEadd(opname, args, invoker, opinv):

Add the arguments -- opname, args, invoker, and opinv into a new element of the queue list. CC module calls this function when the current operation is queued.

4.3 QUEUEdelete(opname, args, invoker, opinv):

Search the operation in the current queue list by using the arguments -- opname, args, and invoker. If the search is successful then it copies the opinv inside the the matched element of queue into the interface argument 'opinv' and returns OK. Otherwise, return ERROR.

4.4 QUEUEprint(index):

Print the useful information from the queue element, queue[index]. The information includes the operation name 'opname', all the arguments 'args' which includes the object name, and the invoker's path name 'invoker'.

5 The Pattern Machines

This is the part for PAT module. It contains a data structure, pattern, for pattern machines and the localized functions for the pattern machine.

5.1 The Data Structure in the PAT Module:

pattern : The data structure is for storing the pattern machines. The data structure is a linked list for the patterns and is global in the PAT module. Each pattern entry (see also in *Figure 6*) contains the following fields.

start_state - the start state

final_states - a list of the final states

dead_state - the only dead state in the pattern

curr_state - the current state

update_state - the destination state after operation invocation is invoked successfully and will be updated when the operation result is returned

(*update_action)() - a pointer that points to a function which returns an integer. And we store the pointer for the update use only

update_patargs - a record array of the arguments of the pointer pointing to functions. Each record has three char strings -- opname, object, and membername.

ptn_trans - a record array for the transitions of the machine. Each record contains an integer begin state, 'begin_state', and a set of transition information. Each transition information contains the following seven fields:

opname -- the char string of the operation name

object -- a char string of the object name

membername -- a char string of the membername

dest_state -- the destination state of the transition

(*loc_action)() -- a pointer points to functions which will update the local variables of the pattern

(*action)() -- a pointer points to functions which will do the action

patargs -- the arguments of the action functions

PATsubs - a record array for storing the subordinate patterns. Each record contains the following four fields:

opname -- a char string of the operation name

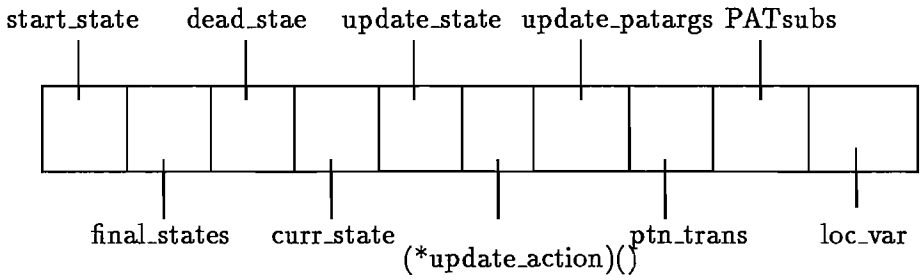
object -- a char string of the object name

membername -- a char string of the membername

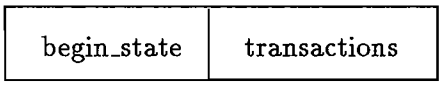
subords -- a list of the subordinate pattern indices

locvar - an array of local variables which are stored as char strings

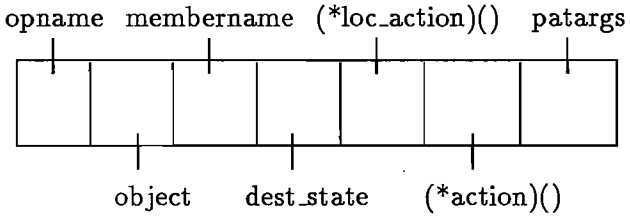
(a) Data structure for an entry of pattern.



(b) Data structure for ptn_trans in part (a).



(c) Data structure for transactions in part (b).



(d) Data structure for PATsubs in part (a).

Figure 6: Data structure for pattern.

5.2 PATinquire(P, opname, args, membername):

CC module calls the function to do the operation invocation and the function returns one of the results, "MATCH", "NOMATCH", and "CONFLICT", back to the CC module.

- compare the operation invocation information with the predicates of transitions where the begin state of the transitions is the particular pattern's current state
- if no proper predicate matches the input opname, object (the first element of the args), membername, and the local action (*loc_action)() returned OK, return NOMATCH back to the CC module
- if there is a predicate which matches the operation, then we return CONFLICT back to the CC module if the destination state is the dead state of the pattern machine. Otherwise, copy the destination state, the actions, and the arguments for these actions into the related update fields and return MATCH back to the CC module.

5.3 PATadd(SP, count):

This function adds the count , count, into the subordinate pattern, pattern[SP].

5.4 PATcreate(template, SP, finder, index, count):

This function creates a new subordinate pattern, pattern[SP], and puts the initial values into the new subordinate pattern including the start state 'start_state', the final states 'final_states', the dead state 'dead_state', the current state 'curr_state', the local variable count in the first element of 'locvar', and the related transitions. This function could be changed by giving different examples of pattern machines.

5.5 PATsubords (P, opname, object, membername, subords):

This function copies the opname, object, membername, and the subords into the data structure 'PATsubs' in the pattern[P].

5.6 PATresult(P, opinv, args, result, inactive):

Pattern pattern[P] receives the opinv's result from the CC module.

- If the result is ERROR, it returns OK and does nothing.
- Performs the actions in the 'update_action' field. This could call functions 'Release()', or 'waste()'. If we get some errors from these functions, ERROR is returned.
- Updates the current state.
- Returns OK.

5.6.1 Release(patargs, args, PATsubs, inactive):

For every operation to be released, this function finds the proper subordinate pattern, sub, and calls function PATrelease(sub) to release the subordinate function and marks it in "inactive" whether the subordinate pattern is inactive or not.

PATrelease(sub): increases the local variable count 'locvar[0]' by 1 and if the variable count = 0, returns INACTIVE. Otherwise returns OK.

5.6.2 waste(patargs, args, PATsubs, inactive):

A waste function which does nothing and always returns OK.

5.7 PATdecr(SP, count):

Decrement the integer count from the local variable 'locvar[0]' field and return INACTIVE if the local variable 'locvar[0]' of the subordinate pattern is zero. Or returns OK otherwise.

5.8 PATfree(P):

Clears the pattern[P] from the memory.

5.9 PATcommit(P):

This function checks whether the current state of the pattern, pattern[P], is in final states. If yes, returns OK. If not, returns FAIL.

6 C Programming Codes

This part contains the C programming codes for the above modules. It includes a make file for the project, four source files (.c files) and three libraries (.h files) in this part as follows.

makefile	: the make file for the project
DRIVER.c	: the source file for the driver part
CC.c	: the source file for the CC module
CC.h	: the library for the CC module
QUEUE.c	: the source file for the QUEUE module
QUEUE.h	: the library for the QUEUE module
PAT.c	: the source file for the PAT module
PAT.h	: the library for the the PAT module

And we list all these codes by using the above order.

```
CFLAGS = -g

CC = cc

SOURCE = DRIVER.c CC.c PAT.c QUEUE.c

OBJECT = DRIVER.o CC.o PAT.o QUEUE.o

HEADER =

EXEC = output

all : $(EXEC)

#clean :
#       rm $(OBJECT)

$(EXEC) : $(OBJECT)
        $(CC) $(CFLAGS) -o $(EXEC) $(OBJECT)

DRIVER.o : DRIVER.c CC.h
        $(CC) $(CFLAGS) -c DRIVER.c

CC.o : CC.c CC.h CCfinder.h PAT.h QUEUE.h
        $(CC) $(CFLAGS) -c CC.c

PAT.o : PAT.c CC.h PAT.h patterns.h
        $(CC) $(CFLAGS) -c PAT.c

QUEUE.o : QUEUE.c QUEUE.h CC.h
        $(CC) $(CFLAGS) -c QUEUE.c
```

```
#include<stdio.h>
#include "CC.h"

ANS result;

main()
{
  int   loop = 1;
  int   option;
  int   tg;
  char  opname[SSIZE];
  char  args[ARGSSIZE][SSIZE];
  char  invoker[PATHSIZE];
  char  opinv[PATHSIZE];

  while (loop) {
        printf("\nThe forms of interaction:\n");
        printf("\t(1) op1 invocation from ct;\n");
        printf("\t(2) op invocation from queue;\n");
        printf("\t(3) op2 invocation from op1;\n");
        printf("\t(4) op result;\n");
        printf("\t(5) TG commit request;\n");
        printf("\t(6) list queue;\n");
        printf("\t(7) list CCinv in CC module; \n");
        printf("\t(8) quit.\n\n");
        printf("Please choose an option number from above [1-8].\n");
        scanf("%d", &option);
        switch (option) {
          case 1 :
          case 2 :
          case 3 :
                GETinput(option, opname, invoker, args);
                CCinvoke(option, opname, args, invoker);
                break;

          case 4 :
                GETresult(opinv);
                CCresult(result, opinv);
                break;

          case 5 :
                printf("Which TG wants to commit?\n");
                scanf("%d", &tg);
                if (CCcommit(tg) == FAIL)
                   printf("\nTG %d commit request fails.\n", tg);
                break;
          case 6 :
                CCqueue_print();
                break;

          case 7 :
                CCinv_print();
                break;

          case 8 :
```

```
                loop = 0;
                break;

          default :
                printf("The input option number was wrong. Try again!!\n");
                break;
        } /* end switch */
  } /* end while */
} /* end main */

/*****************************************************************************/
/*                                                                           */
/*      GETinput(option, opname, invoker, args)                              */
/*                                                                           */
/*****************************************************************************/
long GETinput(option, opname, invoker, args)
int     option;
char    opname[];
char    invoker[];
char    args[][SSIZE];
{
  int   n;
  int   i;      /* for loop controls */

  printf("\nInput the op invocation information:\n");
  printf("\noperation name = \n");
  scanf("%s",opname);
  printf("object name =\n");
  scanf("%s",args[0]);
  printf("Besides object, how many arguments does the %s operation have?\n",
        opname);
  scanf("%d",&n);
  for (i = 1; i <= n; i++) {
        printf("args[%d] =\n",i);
        scanf("%s",args[i]);
  }
  strcpy(args[i],"end");
  if (option == 1)
        printf("ct member name = \n");
  else if (option == 2)
        printf("invoker [a ct member or an object id] =\n");
  else
        printf("invoker's object id =\n");
  scanf("%s", invoker);
}

/*****************************************************************************/
/*                                                                           */
/*      GETresult(opinv)                                                     */
/*                                                                           */
/*****************************************************************************/
long GETresult(opinv)
char opinv[];
{
```

```
    printf("\nInputing the op result information:\n");
    printf("opinv =\n");
    scanf("%s", opinv);
    while (1) {
      printf("Possible types of the op result:\n");
      printf("\t(0) ERROR      ;\n");
      printf("\t(1) integer    ;\n");
      printf("\t(2) float      ;\n");
      printf("\t(3) long       ;\n");
      printf("\t(4) char string;\n");
      printf("\nPlease input the relative type number [0-4] from the menu.\n");
      scanf("%d", &result.r_type);
      switch (result.r_type) {
          case 0 :
             printf("The object result is error.\n");
             return ;
          case 1 :
             printf("Please input the result integer.\n");
             scanf("%d",&result.uval.ival);
             return ;
          case 2 :
             printf("Please input the result float.\n");
             scanf("%f",&result.uval.fval);
             return ;
          case 3 :
             printf("Please input the result long.\n");
             scanf("%ld",&result.uval.lval);
             return ;
          case 4 :
             printf("Please input the result char string.\n");
             scanf("%s",result.uval.sval);
             return ;
          default :
             printf("-- bad type number. Try again!!\n");
             break;
      } /* end switch */
    } /* end while */
}
```

```
#include <stdio.h>
#include "CC.h"
#include "QUEUE.h"
#include "PAT.h"
#include "CCfinder.h"

static int bufnum = 1;          /* the count number for OPinvoke */
static int p_num = 0;           /* the element count number in P */
static INVTYPE CCinv[PSIZE];
static INVTYPE CCinv_last[PSIZE];
static SUBSTYPE CCsubs[SUBSIZE];
static int ccsubs_num = 0;
static int subord = PRIMARY_PNUM;

/*****************************************************************************/
/*                                                                           */
/*      CCinvoke(option, opname, args, invoker)                              */
/*                                                                           */
/*****************************************************************************/
long CCinvoke(option, opname, args, invoker)
int     option;
char    opname[];
char    args[][SSIZE];
char    invoker[];
{
  INVTYPE   template;
  char      opinv[PATHSIZE];
  char      new_opinv[PATHSIZE];  /* the input op's op invocation identifier */
  char      CT[PATHSIZE];
  char      temp[PATHSIZE];
  char      temp2[PATHSIZE];
  int       i;
  int       n;
  int       loop = 1;
  int       j = -1;

  /* ------------------------------------------------- */
  /*  generate a unique operation invocation identifier  */
  /* ------------------------------------------------- */

  if (option == 1)
     strcpy(opinv, "0");
  else if (option == 2) {
     if (!QUEUEdelete(opname, args, invoker, opinv)) {
        printf("Bad op invocation from queue.\n");
        return ;
     }
  } else {
     printf("input the invoker's opinv please\n");
     scanf("%s", opinv);
  }
  sprintf(new_opinv, "%s.%d", opinv, bufnum++);

  /* ---------------------------------------------------------- */
  /*  find the CT that invoked this operation or superoperation  */
```

```
/* ------------------------------------------------------ */

if (!strncmp(invoker, "0", 1)) {
      i = p_num++;
      strcpy(CT, invoker);
} else {
      sprintf(temp2, "%s.end", opinv);
      sscanf(temp2, "0.%d.%s", &n, temp);
      sprintf(temp, "0.%d", n);
      i = 0;
      while (i < p_num && loop)
        if (CCinv[i] != NULL && !strcmp(CCinv[i]->opinv, temp)) {
            if (strcmp(CCinv_last[i]->opinv, opinv)) {
               printf("The invoker's opinv doesn't match!\n");
               return WRONG ;
            } else if (CCinv_last[i]->next_ptr != NULL && option != 2) {
               printf("\nThe previous op invocation was queued.\n");
               return WRONG;
            } else if (strcmp(CCinv_last[i]->args[0], invoker)) {
               printf("\nThe invoker doesn't match the superoperation's object.\n");
               return WRONG;
            }
            strcpy(CT, CCinv[i]->invoker);
            loop = 0;
        } else
            i++;
      if (i == p_num) {
         printf("No such invoker exists!!\n");
         return WRONG ;
      }
}

/* ------------------------------------------------ */
/*  Enter the values into the new entry -- template  */
/* ------------------------------------------------ */

template = (INVTYPE) malloc (sizeof(INV_SP));
strcpy(template->opname, opname);
while (strcmp(args[++j], "end"))
      strcpy(template->args[j], args[j]);
strcpy(template->args[j], "end");
strcpy(template->invoker, invoker);
strcpy(template->opinv, new_opinv);

/* ------------------------------------------------------------------ */
/*  find the indexed patterns in CCfinder by both TG member and object  */
/* ------------------------------------------------------------------ */

if (FINDERreturn(CT, template, i) == WRONG) {
      cfree(template);
      printf("\nERROR during running the FINDER!!\n");
      return WRONG ;
}

/* ------------------------------------------------------------ */
```

```
  /*  Send the operation request to each nonduplicate patterns and  */
  /*    return CONFLICT, MATCH, or NOMATCH                          */
  /* ------------------------------------------------------------- */

  switch (PTNSreturn(template, CT, i)){
        case CONFLICT :
           Suspend(template, opinv, i);
           bufnum--;
           return ;
        case NOMATCH :
        case MATCH :
           break;
  } /* end switch */


  SAVEp(template, i);

  /* ---------------------------------------------- */
  /*  Create subordinate patterrns for suboperation  */
  /* ---------------------------------------------- */

  if (strncmp(invoker, "0", 1))
        Create_Subords(i);
  else
        CCinv_last[i]->create_subs[0] = -1;
  CCinv_last[i]->subords[0] = -1;
}  /* end CCinvoke */

/*************************************************************************/
/*                                                                       */
/*      FINDERreturn(CT, template, i)                                    */
/*                                                                       */
/*************************************************************************/
int FINDERreturn(CT, template, i)
char       CT[];
INVTYPE    template;
int        i;
{
  int   j = 0;
  int   tgnum;                  /* the tg number */
  int   n;                      /* the member number for CCfinder[tgnum] */
  int   loop = 1;                       /* while loop control */
  char  membername[PATHSIZE];   /* the member path name of CCfinder[tgnum] */
  char  temp[PATHSIZE];
  char  temp2[PATHSIZE];

  sprintf(temp2, "%s.end", CT);
  sscanf(temp2,"%d.%d.%s", &tgnum, &n, temp);
  sprintf(membername, "%d.%d", tgnum, n);
  while (loop) {
        template->p_list[j].tg = tgnum;
        strcpy(template->p_list[j].membername, membername);
        if (FINDptns(template, j++, tgnum, i) == WRONG)
            return WRONG;                       /* ERROR */
        if (strcmp(temp,"end")) {
```

```
            tgnum = n;
            sscanf(temp,"%d.%s",&n, temp2);
            strcpy(temp, temp2);
            sprintf(temp2,"%s.%d", membername, n);
            strcpy(membername, temp2);
        }
        else
            loop = 0;
    } /* end while */
    template->p_list[j].tg = -1;
    return OK ;
} /* end FINDERreturn */


/*****************************************************************************/
/*                                                                           */
/*      FINDptns(template, tgindex, tg, pindex)                              */
/*                                                                           */
/*****************************************************************************/
int FINDptns(template, tgindex, tg, pindex)
INVTYPE     template;
int         tgindex;
int         tg;
int         pindex;
{
    int       i = 0;
    int       j = 0;
    int       n = 0;
    int       loop = 1;
    int       k = 0;
    int       dupl = 0;
    int       m;
    int       r;
    int       p;
    INVTYPE   temp;

    /* ---------------------------------------- */
    /*  find the member's location in CCfinder[tg]  */
    /* ---------------------------------------- */

    while (loop)
        if (!strcmp(CCfinder[tg].mem[0].name,"-2") ||
            !strcmp(CCfinder[tg].mem[i].name,"-1"))
            return WRONG ;          /* ERROR */
        else if (strcmp(CCfinder[tg].mem[i].name, template->p_list[tgindex].membername))
            i++;
        else
            loop = 0;

    /* ---------------------------------------- */
    /*  find the object's location in CCfinder[tg]  */
    /* ---------------------------------------- */

    loop = 1;
    while (loop)
```

```
    if (!strcmp(CCfinder[tg].obj[j].name, "-1")) {
        template->p_list[tgindex].duplicate[dupl] = -1;
        template->p_list[tgindex].match[n].pat = -1;
        return OK;
    } else if (strcmp(CCfinder[tg].obj[j].name, template->args[0]))
        j++;
    else
        loop = 0;

  /* ------------------------------------------------- */
  /*  search the match patterns by membername and object  */
  /* ------------------------------------------------- */

  while (CCfinder[tg].mem[i].ptns[k].pat != -1) {
        m = 0;
        while ((r = CCfinder[tg].obj[j].ptns[m]) != -1 &&
                r != CCfinder[tg].mem[i].ptns[k].pat)
                m++;

        /* ------------------------------- */
        /*  check whether duplicate or not  */
        /* ------------------------------- */

        if (r != -1) {
           loop = 1;
           if ((temp = CCinv_last[pindex]) != NULL)
             while (loop) {
                p = 0;
                while (temp->p_list[tgindex].match[p].pat != -1 &&
                        temp->p_list[tgindex].match[p].pat != r)
                   p++;
                if (temp->p_list[tgindex].match[p].pat == -1) {
                   if (temp == CCinv[pindex])
                        break;
                   temp = temp->pre_ptr;
                } else {        /* if duplicate */
                   template->p_list[tgindex].duplicate[dupl++] = r;
                   loop = 0;
                }
             }
           if (loop) {          /* if non-duplicate */
                template->p_list[tgindex].match[n].extend =
                        CCfinder[tg].mem[i].ptns[k].extend;
                template->p_list[tgindex].match[n++].pat = r;
           }
        }
        k++;
  }
  template->p_list[tgindex].duplicate[dupl] = -1;
  template->p_list[tgindex].match[n].pat = -1;
  return OK;
} /* end FINDptns */

/*****************************************************************************/
/*                                                                           */
```

```
/*      PTNSreturn(template, CT, pindex)                                    */
/*                                                                          */
/****************************************************************************/
int PTNSreturn(template, CT, pindex)
INVTYPE     template;
char        CT[];
int         pindex;
{
  int   i = 0;
  int   match_mark = 0;
  int   j;
  int   P;
  int   m;
  int   n;
  int   p_result;

  while (template->p_list[i].tg != -1) {
    j = (m = (n = 0));
    while ((P = template->p_list[i].match[j].pat) != -1) {
        p_result = PATinquire(P, template->opname, template->args,
                template->p_list[i].membername);
        if (p_result == CONFLICT) {
           if (P < PRIMARY_PNUM || !CONFLICTignore(P, pindex, template)) {
                printf("Pattern #%d makes the operation invocation conflict!\n",
                        P);
                return CONFLICT;
           }
        } else if (p_result == MATCH && P < PRIMARY_PNUM) {
                template->p_list[i].match[m].pat = P;
                template->p_list[i].match[m++].extend =
                        template->p_list[i].match[j].extend;
        } else if (p_result == NOMATCH && P < PRIMARY_PNUM) {
                template->p_list[i].nomatch[n].pat = P;
                template->p_list[i].nomatch[n++].extend =
                        template->p_list[i].nomatch[j].extend;
        }
        ++j;
    }
    template->p_list[i].match[m].pat = -1;
    template->p_list[i].nomatch[n].pat = -1;
    if (m > 0)
        match_mark = 1;
    ++i;
  }
  if (match_mark)
    return MATCH;
  else
    return NOMATCH;
}

/****************************************************************************/
/*                                                                          */
/*      CONFLICTignore(P, pindex, template)                                 */
/*                                                                          */
/****************************************************************************/
```

```
int CONFLICTignore(P, pindex, template)
int         P;
int         pindex;
INVTYPE     template;
{
  int           loop;
  int           i = 0;
  int           j = 0;
  int           k = 0;
  int           m = 0;
  int           pat;
  int           n;
  INVTYPE       temp;

  /* ------------------------------- */
  /*  check if op is a suboperation  */
  /* ------------------------------- */

  if (!strncmp(template->invoker, "0", 1))
        return FAIL;

  /* ------------------------------- */
  /*  search the location in CCsubs  */
  /* ------------------------------- */

  loop = 1;
  while (loop) {
        if (i == ccsubs_num) {
            printf("Can't find subord. pattern #%d in CCsubs!!\n", P);
            return FAIL;
        } else if (CCsubs[i]->subname == P)
            loop = 0;
        else
            i++;
  }

  /* ------------------------------------------------------------ */
  /*  check whether the superord. patterns of P are all extend.  */
  /* ------------------------------------------------------------ */

  if (CCsubs[i]->extend == 0)
        return FAIL;

  /* ------------------------------------------------------------------------ */
  /*  check if all the superordinate patterns of the subordinate pattern      */
  /*    P match with one of the superoperations of the operation              */
  /* ------------------------------------------------------------------------ */

  while (CCsubs[i]->tg != CCinv_last[pindex]->p_list[j].tg)
        j++;
  while (strcmp(CCsubs[i]->opsub[k].opinv, "-1")) {
        n = 0;
        while ((pat = CCsubs[i]->opsub[k].superords[n]) != -1) {
                temp = CCinv_last[pindex];
                while (1) {
```

```
                    loop = 1;
                    m = 0;
                    while (loop && temp->p_list[j].match[m].pat != -1)
                        if (temp->p_list[j].match[m++].pat == pat)
                                loop = 0;
                    m = 0;
                    while  (loop && temp->p_list[j].nomatch[m].pat != -1)
                        if (temp->p_list[j].nomatch[m++].pat == pat)
                                loop = 0;
                    if (loop && temp != CCinv[pindex])
                        temp = temp->pre_ptr;
                    else if (loop)
                        return FAIL;
                    else {
                        n++;
                        break;
                    }
                }
        }
        k++;
  }
  return OK;
}


/*****************************************************************************/
/*                                                                           */
/*      Suspend(template, opinv, i)                                          */
/*                                                                           */
/*****************************************************************************/
long Suspend(template, opinv, i)
INVTYPE    template;
char       opinv[];
int        i;
{
  printf("\nThe '%s' op invocation makes the patterns conflict!!\n",
        template->opname);
  QUEUEadd(template->opname, template->args, template->invoker, opinv);
  if (!strncmp(template->invoker, "0", 1))
        p_num--;
  else
        CCinv_last[i]->next_ptr = CCinv_last[i];
  cfree(template);
}

/*****************************************************************************/
/*                                                                           */
/*      SAVEp(template, pindex)                                              */
/*                                                                           */
/*****************************************************************************/
long SAVEp(template, pindex)
INVTYPE    template;
int        pindex;
{
```

```
    char    temp[PATHSIZE];

    template->next_ptr = NULL;
    if (!strncmp(template->invoker, "0", 1)) {
          template->pre_ptr = NULL;
          CCinv[pindex] = template;
    } else {
          CCinv_last[pindex]->next_ptr = template;
          template->pre_ptr = CCinv_last[pindex];
    }
    CCinv_last[pindex] = template;
}

/*****************************************************************************/
/*                                                                           */
/*      Create_Subords(ccindex)                                              */
/*                                                                           */
/*****************************************************************************/
long Create_Subords(ccindex)
int     ccindex;
{
    INVTYPE         temp;
    SUBSTYPE        temp_sub;
    int             i = 0;
    int             sub = 0;
    int             j;
    int             k;
    int             m;
    int             n;
    int             p;
    int             q;
    int             tg;
    int             count;
    int             superpat;
    int             loop;
    int             is_superpat;
    int             pat;
    int             addsub[10];
    int             ind;

    /* ------------------------------------------------------------------------ */
    /*  for each tg in the p_list of the operation, if it's superoperations' matched  */
    /*    patterns are non-duplicate with this operation, then add a new field in     */
    /*    CCsubs if there is one or create a new one if no such entry exists          */
    /* ------------------------------------------------------------------------ */

    while ((tg = CCinv_last[ccindex]->p_list[i].tg) != -1) {
       temp_sub = (SUBSTYPE) malloc (sizeof(SUB_SP));
       temp_sub->extend = 1;
       count = 0;
       temp = CCinv_last[ccindex]->pre_ptr;
       addsub[(ind = 0)] = 0;
       while (temp != NULL) {
          j = 0;
          while ((superpat = temp->p_list[i].match[j].pat) != -1) {
```

```
        is_superpat = (loop = 1);
        k = 0;
        while (CCinv_last[ccindex]->p_list[i].duplicate[k] != -1 && loop)
             if (CCinv_last[ccindex]->p_list[i].duplicate[k] == superpat)
                is_superpat = (loop = 0);
             else
                k++;
        if (is_superpat) {
             temp_sub->opsub[1].superords[count++] = superpat;
             if (temp->p_list[i].match[j].extend == 0)
                temp_sub->extend = 0;
             addsub[ind] = 1;
        }
        j++;
     }
     temp = temp->pre_ptr;
     addsub[++ind] = 0;
  }

  /* ---------------------------------------------------------------- */
  /*  if the superord patterns in temp_sub exists (i.e. count > 0),   */
  /*     then entry a new space in CCsubs for them.                   */
  /* ---------------------------------------------------------------- */

  if (count == 0)
     cfree(temp_sub);
  else {
     temp_sub->opsub[1].superords[count] = -1;
     m = 0;
     while (m < ccsubs_num) {
        if (!strcmp(CCinv_last[ccindex]->args[0], CCsubs[m]->object) &&
             !strcmp(CCinv_last[ccindex]->opname, CCsubs[m]->opname) &&
             !strcmp(CCinv[ccindex]->invoker, CCsubs[m]->CT) &&
             CCinv_last[ccindex]->p_list[i].tg == CCsubs[m]->tg) {

           p = -1;
           loop = 1;
           while (loop)
              if (!strcmp(CCsubs[m]->opsub[++p].opinv, "-1")) {
                 strcpy(CCsubs[m]->opsub[p+1].opinv, "-1");
                 loop = 0;
              } else if (!strcmp(CCsubs[m]->opsub[p].opinv, "-2"))
                 loop = 0;
           j = 0;
           q = 0;
           while ((superpat = temp_sub->opsub[1].superords[j++]) != -1) {
              loop = 1;
              n = -1;
              while (loop && strcmp(CCsubs[m]->opsub[++n].opinv, "-1")) {
                 if (strcmp(CCsubs[m]->opsub[n].opinv, "-2")) {
                    k = 0;
                    while (loop && CCsubs[m]->opsub[n].superords[k] != -1)
                      if (CCsubs[m]->opsub[n].superords[k++] == superpat) {
                             count--;
                             loop = 0;
```

```
                    }
                }
            }
            if (loop)
                CCsubs[m]->opsub[p].superords[q++] = superpat;
        }
        if (count > 0) {
            CCsubs[m]->opsub[p].superords[q] = -1;
            strcpy(CCsubs[m]->opsub[p].opinv, CCinv_last[ccindex]->opinv);
            if (CCsubs[m]->extend && !temp_sub->extend)
                    CCsubs[m]->extend = 0;
            PATadd(CCsubs[m]->subname, count);
            printf("\nadd a new subord field in CCsub[%d]\n", m);
        }
        cfree(temp_sub);
        break;
    } else
        m++;
}

if ( m == ccsubs_num) {
    temp_sub->subname = subord;
    temp_sub->tg = tg;
    strcpy(temp_sub->opname, CCinv_last[ccindex]->opname);
    strcpy(temp_sub->object, CCinv_last[ccindex]->args[0]);
    strcpy(temp_sub->CT, CCinv[ccindex]->invoker);
    strcpy(temp_sub->opsub[0].opinv, "complete");
    temp_sub->opsub[0].superords[0] = -1;
    strcpy(temp_sub->opsub[2].opinv, "-1");
    CCsubs[ccsubs_num++] = temp_sub;
    PATcreate(CCinv_last[ccindex], subord, CCfinder[tg].mem, i,
        count);
    printf("\ncreate a new subord pattern %d in CCsubs[%d]\n", subord,
        m);

    /* ---------------------------------------- */
    /*  add the new subord pattern into CCfinder  */
    /* ---------------------------------------- */

    j = 0;
    while (strcmp(CCfinder[tg].mem[j].name, "-1")) {
      if (strcmp(CCfinder[tg].mem[j].name, "-3")) {
        k = 0;
        while ((pat = CCfinder[tg].mem[j].ptns[k].pat) != -1 &&
             pat != -2)
                k++;
        CCfinder[tg].mem[j].ptns[k].pat = subord;
        CCfinder[tg].mem[j].ptns[k].extend = CCsubs[m]->extend;
        if (pat == -1)
             CCfinder[tg].mem[j].ptns[++k].pat = -1;
      }
      j++;
    }
    loop = 1;
    j = 0;
```

```
            while (loop) {
                 if (!strcmp(CCfinder[tg].obj[j].name, "-1")) {
                   strcpy(CCfinder[tg].obj[j].name, CCinv_last[ccindex]->args[0]);
                   CCfinder[tg].obj[j].ptns[0] = subord;
                   CCfinder[tg].obj[j].ptns[1] = -1;
                   strcpy(CCfinder[tg].obj[j+1].name, "-1");
                   loop = 0;
                 } else if (!strcmp(CCfinder[tg].obj[j].name,
                         CCinv_last[ccindex]->args[0])){
                   k = 0;
                   while ((pat = CCfinder[tg].obj[j].ptns[k]) != -1 &&
                         pat != -2)
                         k++;
                   CCfinder[tg].obj[j].ptns[k] = subord;
                   if (pat == -1)
                      CCfinder[tg].obj[j].ptns[++k] = -1;
                   loop = 0;
                 } else
                   j++;
            }
            subord++;
          }


          /* ------------------------------------------------ */
          /*  add the subord pattern to the superop of the op  */
          /* ------------------------------------------------ */

          CCinv_last[ccindex]->create_subs[sub++] = m;
          temp = CCinv_last[ccindex]->pre_ptr;
          ind = -1;
          while (temp != NULL) {
               if (addsub[++ind] == 1) {
                  j = 0;
                  while (temp->subords[j] != -1 && temp->subords[j] != -2)
                     j++;
                  if (temp->subords[j] == -1)
                     temp->subords[++j] = -1;
                  temp->subords[j-1] = m;
               }
               temp = temp->pre_ptr;
          }

      }
      i++;
    }
    CCinv_last[ccindex]->create_subs[sub] = -1;
}

/*****************************************************************************/
/*                                                                           */
/*      CCresult(result, opinv)                                              */
/*                                                                           */
/*****************************************************************************/
long CCresult(result, opinv)
```

```
ANS     result;
char    opinv[];
{
  INVTYPE       template;
  int           i = 0;
  int           j = 0;
  int           k;
  int           P;
  int           q;
  int           loop = 1;
  int           sub;
  int           tg;
  int           m;
  int           objnum;
  int           super;
  int           p;
  int           r;
  int           n;
  int           s;
  int           subords[SUBOBJSIZE][SUBSIZE];
  struct inacttype inactive[SUBOBJSIZE];

  /* --------------------------- */
  /*  search the index in CCinv  */
  /* --------------------------- */

  while (i < p_num && loop)
    if (CCinv_last[i] != NULL && !strcmp(CCinv_last[i]->opinv, opinv))
        loop = 0;
    else
        i++;
  if (i == p_num) {
        printf("\nBad op choice to return the result!\n");
        return ;
  }
  template = CCinv_last[i];

  /* ------------------------------ */
  /*  check if the result is error  */
  /* ------------------------------ */

  if (result.r_type == ERROR) {
        Abort(result, i, opinv);
        return;
  }

  while ((tg = CCinv_last[i]->p_list[j].tg) != -1) {
     objnum = 0;
     k = -1;
     while ((P = template->p_list[j].match[++k].pat) != -1) {
        q = 0;
        while ((sub = template->subords[q++]) != -1) {
           if (sub != -2 && CCsubs[sub]->tg == tg) {
              m = 0;
              while ((super = CCsubs[sub]->opsub[0].superords[m]) != -1)
```

```
            if (super == P)
                  break;
            else
                  m++;
         if (super != -1) {
            m = 0;
            while (m < objnum)
             if (!strcmp(inactive[m].object, CCsubs[sub]->object)) {
                  p = -1;
                  while (inactive[m].index[++p] != -1)
                      ;
                  inactive[m].index[p] = sub;
                  subords[m][p] = CCsubs[sub]->subname;
                  inactive[m].index[p+1] = -1;
                  break;
             } else m++;
            if (m == objnum) {
                  objnum++;
                  strcpy(inactive[m].object, CCsubs[sub]->object);
                  inactive[m].index[0] = sub;
                  inactive[m].index[1] = -1;
                  subords[m][0] = CCsubs[sub]->subname;
            }
        }
      }
   } /* end while */
   m = 0;
   while (m < objnum)
      PATsubords(P, template->opname, inactive[m].object,
           template->p_list[j].membername, subords[m++]);
   if (!PATresult(P, opinv, template->args, result, inactive)) {
      printf("\nNotify object %s that pattern #%d fails to return ",
           template->args[0], P);
      printf("the result!!\n");
      Abort(result, i, opinv);
      return;
   }
   m = 0;
   while (m < objnum) {
      p = 0;
      while ((sub = inactive[m].index[p]) != -1) {
        if (inactive[m].inact[p] == 1)
           Destroy(sub, 0, 1);
        else {
           /* copy field to complete */
           r = 0;
           while (CCsubs[sub]->opsub[0].superords[r] != -1)
                r++;
           n = 0;
           while (strcmp(CCsubs[sub]->opsub[++n].opinv, opinv)) ;
           s = 0;
           while (CCsubs[sub]->opsub[n].superords[s] != -1)
                CCsubs[sub]->opsub[0].superords[r++] =
                   CCsubs[sub]->opsub[n].superords[s++];
           strcpy(CCsubs[sub]->opsub[n].opinv, "-2");
```

```
                CCsubs[sub]->opsub[0].superords[s] = -1;
              }
              p++;
            }
            m++;
          }
        }
        j++;
    }

    /* ------------------------------------ */
    /*  send the result back to the invoker  */
    /* ------------------------------------ */

    RESULTback(result, template->invoker);

    /* ---------------------------------- */
    /*  clear the CCinv entry for opinv  */
    /* ---------------------------------- */

    if (!strncmp(CCinv_last[i]->invoker, "0", 1)) {
          CCinv_last[i] = NULL;
          CCinv[i] = NULL;
    } else {
          CCinv_last[i] = CCinv_last[i]->pre_ptr;
          CCinv_last[i]->next_ptr = NULL;
    }
    cfree(template);
}

/*****************************************************************************/
/*                                                                           */
/*      Abort(result, i, opinv)                                              */
/*                                                                           */
/*****************************************************************************/
long Abort(result, i, opinv)
ANS     result;
int     i;
char    opinv[];
{
    int                     j = 0;
    int                     k;
    int                     P;
    int                     loop;
    int                     sub;
    INVTYPE                 template;
    struct inacttype        waste[SUBOBJSIZE];

    /* ------------------------------ */
    /*  notify each matched pattern  */
    /* ------------------------------ */

    result.r_type = ERROR;
    while (CCinv_last[i]->p_list[j].tg != -1) {
       k = 0;
```

```
        while ((P = CCinv_last[i]->p_list[j].match[k++].pat) != -1)
            if (P != -2)
                PATresult(P, CCinv_last[i]->opinv, CCinv_last[i]->args, result,
                    waste);
        j++;
    }

    /* ------------------------------------------------------------------- */
    /*  for each subord pattern, sub, that opinv created, call Destroy(sub)  */
    /* ------------------------------------------------------------------- */

    j = 0;
    while ((sub = CCinv_last[i]->create_subs[j++]) != -1)
          Destroy(sub, 1, i);

    /* ---------------------------------------------- */
    /*  send the op request back to the invoker as failed  */
    /* ---------------------------------------------- */

    RESULTback(result, CCinv_last[i]->invoker);

    /* ---------------------------------- */
    /*  clear CCinv of the opinv's entry  */
    /* ---------------------------------- */

    template = CCinv_last[i];
    if (!strncmp(CCinv_last[i]->invoker, "0", 1)) {
          CCinv_last[i] = NULL;
          CCinv[i] = NULL;
    } else {
          CCinv_last[i] = CCinv_last[i]->pre_ptr;
          CCinv_last[i]->next_ptr = NULL;
    }
    cfree(template);
}

/*****************************************************************************/
/*                                                                           */
/*      Destroy(sub, option, index)                                          */
/*                                                                           */
/*****************************************************************************/
long Destroy(sub, option, index)
int     sub;
int     option;
int     index;
{
    int       i = 0;
    int       count = 0;
    int       j = 0;
    int       find = 0;
    int       subname;
    int       k;
    int       pat;
    int       tg;
    INVTYPE   template;
```

```
if (option == 1) {
      while (strncmp(CCsubs[sub]->opsub[i].opinv, CCinv_last[index]->opinv,
              strlen(CCinv_last[index]->opinv)))
              i++;
      while (CCsubs[sub]->opsub[i].superords[j] != -1)
              count++;
      strcpy(CCsubs[sub]->opsub[i].opinv, "-2");
      if (!PATdecr(CCsubs[sub]->subname, count))
              option = 0;
}

if (option == 0) {
      /* ---------------------------------------- */
      /*  remove the subord pattern from memory  */
      /* ---------------------------------------- */

      PATfree(CCsubs[sub]->subname);

      /* ------------------------------------------- */
      /*  clear the references to sub from CCfinder  */
      /* ------------------------------------------- */

      while (strcmp(CCfinder[(tg = CCsubs[sub]->tg)].mem[j].name, "-1"))
         if (strcmp(CCfinder[tg].mem[j].name, "-3")) {
              k = 0;
              while (CCfinder[tg].mem[j].ptns[k].pat != subname)
                   k++;
              CCfinder[tg].mem[j++].ptns[k].pat = -2;
         }

      j = 0;
      while (strcmp(CCfinder[tg].obj[j].name, CCsubs[sub]->object))
              j++;
      k = 0;
      while ((pat = CCfinder[tg].obj[j].ptns[k]) != -1)
         if (pat == subname) {
              CCfinder[tg].obj[j].ptns[k] = -2;
              find = 1;
         } else if (pat != -2 && find)
              break;
         else
              k++;
      if (pat == -1)
         strcpy(CCfinder[tg].obj[j].name, "-2");
}

/* ------------------------------------------------------------- */
/*  for each suprop, clear sub's name from superop's CCinv.subords  */
/* ------------------------------------------------------------- */

template = CCinv_last[index]->pre_ptr;
while (template != NULL) {
      i = 0;
      while (template->subords[i] != -1)
```

```
            if (template->subords[i] == sub) {
                template->subords[i] = -2;
                break;
            } else
                i++;
        template = template->pre_ptr;
    }

    /* ---------------------------- */
    /*  clear sub's entry in CCsubs  */
    /* ---------------------------- */

    if (option == 0) {
        cfree(CCsubs[sub]);
        CCsubs[sub] = NULL;
    }
}

/*****************************************************************************/
/*                                                                           */
/*      RESULTback(result, invoker)                                          */
/*                                                                           */
/*****************************************************************************/
long RESULTback(result, invoker)
ANS     result;
char    invoker[];
{
    int   i;

    printf("\nReturn the ");
    if ((i = result.r_type) == ERROR)
        printf("'error' result ");
    else if (i == 1)
        printf("integer '%d' ", result.uval.ival);
    else if (i == 2)
        printf("float '%f' ", result.uval.fval);
    else if (i == 3)
        printf("long '%ld' ", result.uval.lval);
    else
        printf("char string '%s' ", result.uval.sval);
    printf("back to the invoker '%s'.\n", invoker);
}

/*****************************************************************************/
/*                                                                           */
/*      CCcommit(tg)                                                         */
/*                                                                           */
/*****************************************************************************/
int CCcommit(tg)
int     tg;
{
    int   i = 0;
    int   subtg;
    int   j;
    int   P;
```

```
int   supertg;
int   loop;
char  member[PATHSIZE];
char  temp[PATHSIZE];

if (!strcmp(CCfinder[tg].mem[0].name, "-2") ||
    !strcmp(CCfinder[tg].mem[0].name, "-1") ||
    !strcmp(CCfinder[tg].mem[0].name, "")) {
      printf("Bad TG choice to commit!!\n");
      return FAIL;
}

while (strcmp(CCfinder[tg].mem[i].name, "-1")) {
   if (strcmp(CCfinder[tg].mem[i].name, "-3")) {
      strcpy(member, CCfinder[tg].mem[i].name);
      sscanf(rindex(member, '.'), ".%d", &subtg);
      j = 0;
      loop = 1;
      while (loop) {
         if (!strcmp(CCfinder[subtg].mem[j].name, "-1") ||
              !strcmp(CCfinder[subtg].mem[j].name, ""))
              loop = 0;
         else if (strcmp(CCfinder[subtg].mem[j].name, "-3")) {
              sprintf(temp, "%s.", member);
              if (!strncmp(CCfinder[subtg].mem[j].name, temp,
                      strlen(temp)))
                  return FAIL;
              loop = 0;
         } else
              j++;
      }
      j = 0;
      while ((P = CCfinder[tg].mem[i].ptns[j].pat) != -1) {
         if (P != -2)
              if (!PATcommit(P))
                      return FAIL;
         j++;
      }
   }
   i++;
} /* end while */

loop = 1;
while (loop) {
   printf("\nTG %d has already commited.\n", tg);
   strcpy(CCfinder[tg].mem[0].name, "-1");
   if (tg == 0)
      return OK;
   strcpy(temp, member);
   temp[strlen(member) - strlen(rindex(member, '.'))] = '\0';
   strcpy(member, temp);
   member[strlen(temp) - strlen(rindex(temp, '.'))] = '\0';
   if (!strcmp(member, "0"))
      supertg = 0;
   else
```

```
        sscanf(rindex(member, '.'), ".%d", &supertg);
      j = 0;
      i = 0;
      while (strcmp(CCfinder[supertg].mem[j].name, "-1")) {
        if (!strcmp(CCfinder[supertg].mem[j].name, temp)) {
          i = 1;
          strcpy(CCfinder[supertg].mem[j].name, "-3");
        } else if (strcmp(CCfinder[supertg].mem[j].name, "-3"))
          loop = 0;
        if (i && !loop)
          break;
        j++;
      }
      tg = supertg;
      strcpy(member, temp);
    }
    return OK;
}

/*****************************************************************************/
/*                                                                           */
/*      CCqueue_print()                                                      */
/*                                                                           */
/*****************************************************************************/
long CCqueue_print()
{
  int   i = 0;

  printf("\nThe waiting operations are the following:\n\n");
  while (1)
    if (!QUEUEprint(i++)) break;
  printf("\nFinsih listing the queue!\n");
}

/*****************************************************************************/
/*                                                                           */
/*      CCinv_print()                                                        */
/*                                                                           */
/*****************************************************************************/
long CCinv_print()
{
  int       i = 0;
  int       j;
  int       k;
  INVTYPE   temp;

  while (i < p_num) {
    if (CCinv[i] != NULL) {
        temp = CCinv[i];
        printf("\nCCinv[%d] : \n", i);
        while (temp != NULL) {
                printf("\topname = %s,\n", temp->opname);
                printf("\targs[] = ");
                j = -1;
                while (strcmp(temp->args[++j], "end"))
```

```
                printf("%s,\t", temp->args[j]);
            printf("\n\tinvoker = %s,\n\topinv = %s,\n\tp_list :\n ",
                    temp->invoker, temp->opinv);
            j = 0;
            while (temp->p_list[j].tg != -1) {
                printf("\t  tg : %d  --  plist : ", temp->p_list[j].tg);
                k = 0;
                while (temp->p_list[j].match[k].pat != -1)
                    printf("%d, ", temp->p_list[j].match[k++].pat);
                printf("\n");
                ++j;
            }
            printf("\n\t==============================\n\n");
            if (temp == CCinv_last[i])
                    temp = NULL;
            else
                    temp = temp->next_ptr;
        }
    }
    i++;
  }
}
```

```
#define ARGSSIZE        10      /* args size  */
#define ASIZE           30      /* array size */
#define PSIZE           20      /* CCinv size */
#define PATHSIZE        30      /* path name size */
#define SSIZE           15      /* string size */
#define NAMESIZE        20
#define PTNSSIZE        50
#define MEMSIZE         10
#define OBJSIZE         20
#define TGSIZE          10
#define OPSIZE          10

#define OK              1
#define FAIL            0

#define ERROR           0
#define WRONG           -3

typedef struct {
   int  pat;
   int  extend;
}EXTYPE;

typedef struct {
   struct memtype{
        char    name[NAMESIZE];
        EXTYPE  ptns[PTNSSIZE];
   } mem[MEMSIZE];

   struct {
        char    name[NAMESIZE];
        int     ptns[PTNSSIZE];
   } obj[OBJSIZE];
}FINDER;

typedef struct oppl * INVTYPE;

typedef struct oppl{
   INVTYPE      next_ptr;
   INVTYPE      pre_ptr;
   char         opname[SSIZE];
   char         args[ARGSSIZE][SSIZE];
   char         invoker[PATHSIZE];
   char         opinv[PATHSIZE];
   int          subords[PTNSSIZE];
   int          create_subs[PTNSSIZE];
   struct plist_type {
      int       tg;
      int       duplicate[ASIZE];
      char      membername[PATHSIZE];
      EXTYPE    match[ASIZE];
      EXTYPE    nomatch[ASIZE];
   }p_list[TGSIZE];
}INV_SP;
```

```
typedef struct {
   int          subname;
   int          tg;
   int          extend;
   char         opname[SSIZE];
   char         object[SSIZE];
   char         CT[PATHSIZE];
   struct subinfo {
     int        superords[PTNSSIZE];
     char       opinv[PATHSIZE];
   } opsub[OPSIZE];
}SUB_SP, *SUBSTYPE;

typedef struct {
   int  r_type;
   union rval {
      int       ival;
      float     fval;
      long      lval;
      char      sval[SSIZE];
   }uval;
}ANS;


extern long CCinvoke();
extern long CCresult();
extern int  CCcommit();
extern long CCqueue_print();
extern long CCinv_print();
```

```
#include<stdio.h>
#include "CC.h"
#include "QUEUE.h"

static qnum = -1;
static QTYPE queue[QUEUESIZE];

/*****************************************************************************/
/*                                                                           */
/*      QUEUEadd(opname, args, invoker, opinv)                               */
/*                                                                           */
/*****************************************************************************/
long QUEUEadd(opname, args, invoker, opinv)
char    opname[];
char    args[][SSIZE];
char    invoker[];
char    opinv[];
{
   int     i = 0;               /* for counting args */
   QTYPE   qpiece;

   qpiece = (queue[++qnum] = (QTYPE) malloc (sizeof (Q_SP)));
   strcpy(qpiece->opname, opname);
   while (strcmp(args[i], "end"))
        strcpy(qpiece->args[i], args[i++]);
   strcpy(qpiece->args[i],"end");
   strcpy(qpiece->invoker, invoker);
   strcpy(qpiece->opinv, opinv);
}

/*****************************************************************************/
/*                                                                           */
/*      QUEUEdelete(opname, args, invoker, opinv)                            */
/*                                                                           */
/*****************************************************************************/
int QUEUEdelete(opname, args, invoker, opinv)
char    opname[];
char    args[][SSIZE];
char    invoker[];
char    opinv[];
{
   int    next;                 /* while loop control  */
   int    i;                    /* for counting args[] */
   int    index = qnum;

   while (index >= 0)  {
     if (queue[index] != NULL)
        if (!strcmp(queue[index]->opname, opname))
          if (!strcmp(queue[index]->invoker, invoker)) {
             i = 0;
             next = 1;
             while (next)
                if (!strcmp(args[i], "end")) {
                    next = 0;
                    if (!strcmp(queue[index]->args[i],"end")) {
```

```
                    strcpy(opinv, queue[index]->opinv);
                    cfree(queue[index]);
                    queue[index] = NULL;
                    return OK;
                }
            }
            else if (!strcmp(queue[index]->args[i], args[i]))
                i++;
            else
                next = 0;
        }
    index--;
  }
  return ERROR;
}

/*****************************************************************************/
/*                                                                           */
/*      QUEUEprint(index)                                                    */
/*                                                                           */
/*****************************************************************************/
long QUEUEprint(index)
int     index;
{
  int       i = 1;              /* for counting args */
  QTYPE     qpiece;

  if (index > qnum) return ERROR;
  else if (queue[index] == NULL) return OK;
  else qpiece = queue[index];

  printf("\topname = %s\n", qpiece->opname);
  printf("\tobject = %s\n", qpiece->args[0]);
  while (strcmp(qpiece->args[i], "end")) {
        printf("\targs[%d] = %s\n", i, qpiece->args[i]);
        i++;
  }
  printf("\tinvoker = %s\n", qpiece->invoker);
  printf("\t=========================\n\n");
  return OK;
}
```

```
#define QUEUESIZE       100     /* max size of the queue list */

typedef struct {
  char  opname[SSIZE];
  char  args[ARGSSIZE][SSIZE];
  char  invoker[PATHSIZE];
  char  opinv[PATHSIZE];
}Q_SP, *QTYPE;

extern int  QUEUEadd();
extern int  QUEUEdelete();
extern long QUEUEprint();
```

```
#include <stdio.h>
#include "CC.h"
#include "PAT.h"
#include "patterns.h"

/*****************************************************************************/
/*                                                                           */
/*      PATinquire(P, opname, args, membername)                              */
/*                                                                           */
/*****************************************************************************/
int PATinquire(P, opname, args, membername)
int     P;
char    opname[];
char    args[][SSIZE];
char    membername[];
{
  PTNTYPE       pat = pattern[P];
  struct ttype  temp;
  int           i = 0;

  while (1) {
     temp = pat->ptn_trans[pat->curr_state].transitions[i];
     if (!strcmp(temp.opname, "-1"))
           return NOMATCH;
     else if (!strcmp(temp.opname, opname) && !strcmp(temp.object, args[0]) &&
                !strcmp(temp.membername, membername) &&
                (*(temp.loc_action))(args, pat->locvar) == OK) {
           if (pat->dead_state == temp.dest_state)
                return CONFLICT;
           pat->update_state = temp.dest_state;
           pat->update_action = temp.action;
           strcpy(pat->update_patargs, temp.patargs);
           return MATCH;
     } else
           i++;
  }
}

/*****************************************************************************/
/*                                                                           */
/*      PATadd(SP, count)                                                    */
/*                                                                           */
/*****************************************************************************/
long PATadd(SP, count)
int     SP;
int     count;
{
  int           value;
  PTNTYPE       subpat = pattern[SP];

  value = atoi(subpat->locvar[0]) + count;
  sprintf(subpat->locvar[0], "%d", value);
}

/*****************************************************************************/
```

```
/*                                                                          */
/*      PATcreate(template, SP, finder, index, count)                       */
/*                                                                          */
/**************************************************************************/
long PATcreate(template, SP, finder, index, count)
INVTYPE         template;
int             SP;
struct memtype  finder[];
int             index;
int             count;
{
  PTNTYPE       subpat = pattern[SP];
  int           i = 0;
  int            j = 0;
  char          member[PATHSIZE];

  subpat = (PTNTYPE) malloc (sizeof(P_SP));
  subpat->start_state = 0;
  subpat->dead_state = 2;
  subpat->curr_state = 1;
  subpat->update_state = 1;
  sprintf(subpat->locvar[0], "%d", count);
  subpat->final_states[0] = 0;
  subpat->final_states[1] = 1;
  subpat->final_states[2] = -1;
  subpat->ptn_trans[1].begin_state = 1;
  while (strcmp(finder[i].name, "-1")) {
     strcpy(member, finder[i++].name);
     if (!strcmp(member, "-2"))
        break;
     else if (!strcmp(member, template->p_list[index].membername)) {
        subpat->ptn_trans[0].begin_state = 0;
        strcpy(subpat->ptn_trans[0].transitions[0].opname, template->opname);
        strcpy(subpat->ptn_trans[0].transitions[0].object, template->args[0]);
        strcpy(subpat->ptn_trans[0].transitions[0].membername, member);
        subpat->ptn_trans[0].transitions[0].dest_state = 1;
        strcpy(subpat->ptn_trans[0].transitions[1].opname, "-1");
     } else {
        strcpy(subpat->ptn_trans[1].transitions[j].opname, "W");
        strcpy(subpat->ptn_trans[1].transitions[j].object, template->args[0]);
        strcpy(subpat->ptn_trans[1].transitions[j].membername, member);
        subpat->ptn_trans[1].transitions[j++].dest_state = 2;
     }
  }
  strcpy(subpat->ptn_trans[1].transitions[j].opname, "-1");
}

/**************************************************************************/
/*                                                                          */
/*      PATsubords(P, opname, object, membername, subords)                  */
/*                                                                          */
/**************************************************************************/
long PATsubords(P, opname, object, membername, subords)
int     P;
char    opname[];
```

```
char    object[];
char    membername[];
int     subords[SUBSIZE];
{
  PTNTYPE       pat = pattern[P];
  int           i = 0;
  int           j = 0;

  while (strcmp(pat->PATsubs[i].opname, "-1"))
        i++;
  strcpy(pat->PATsubs[i].opname, opname);
  strcpy(pat->PATsubs[i].object, object);
  strcpy(pat->PATsubs[i].membername, membername);
  while (subords[j] != -1)
        pat->PATsubs[i].subords[j] = subords[j++];
  pat->PATsubs[i].subords[j] = -1;
  strcpy(pat->PATsubs[i++].opname, "-1");
}

/*****************************************************************************/
/*                                                                           */
/*      PATresult(P, opinv, args, result, inactive)                          */
/*                                                                           */
/*****************************************************************************/
int PATresult(P, opinv, args, result, inactive)
int                 P;
char                opinv[];
char                args[][SSIZE];
ANS                 result;
struct inacttype    inactive[];
{
  PTNTYPE       pat = pattern[P];
  int           i;

  if (result.r_type == ERROR)
        return OK;

  if ((*(pat->update_action))(pat->update_patargs, args, pat->locvar,
                pat->PATsubs, inactive)) {
        pat->curr_state = pat->update_state;
        printf("The new current state of the pattern #%d is state %d.\n",
                 P, pat->curr_state);
        return OK;
  } else
        return ERROR;
}

/*****************************************************************************/
/*                                                                           */
/*      Release(patargs, args, loc_var, PATsubs, inactive)                   */
/*                                                                           */
/*****************************************************************************/
int Release(patargs, args, loc_var, PATsubs, inactive)
struct ptn_varr         patargs[];
char                    args[][SSIZE];
```

```
char                    loc_var[][SSIZE];
struct subtype          PATsubs[];
struct inacttype        inactive[];
{
  int       i = 0;
  int       j;
  int       m;
  int       sub;

  while (strcmp(patargs[i].opname, "-1")) {
     j = 0;
     while (strcmp(PATsubs[j].opname, "-1")) {
        if (!strcmp(patargs[i].opname, PATsubs[j].opname) &&
            !strcmp(patargs[i].object, PATsubs[j].object) &&
            !strcmp(patargs[i].membername, PATsubs[j].membername)) {
                m = 0;
                while ((sub = PATsubs[j].subords[m]) != -1) {
                    if (!PATrelease(pattern[sub]))
                        inactive[j].inact[m] = 1;
                    else
                        inactive[j].inact[m] = 0;
                    m++;
                }
        }
        j++;
     }
     i++;
  }
  return OK;
}

/*****************************************************************************/
/*                                                                           */
/*      waste()                                                              */
/*                                                                           */
/*****************************************************************************/
int waste()
{
  return OK;
}

/*****************************************************************************/
/*                                                                           */
/*      PATrelease(sub)                                                      */
/*                                                                           */
/*****************************************************************************/
int PATrelease(sub)
PTNTYPE     sub;
{
  int    count = atoi(sub->locvar[0]);

  if ((count -= 1) == 0)
        return INACTIVE;
  else {
        sprintf(sub->locvar[0], "%d", count);
```

```
        return OK;
    }
}

/*****************************************************************************/
/*                                                                           */
/*      PATdecr(SP, count)                                                   */
/*                                                                           */
/*****************************************************************************/
int PATdecr(SP, count)
int     SP;
int     count;
{
  PTNTYPE       sub = pattern[SP];
  int           value = atoi(sub->locvar[0]);

  if ((value -= count) == 0)
        return INACTIVE;
  else {
        sprintf(sub->locvar[0], "%d", value);
        return OK;
  }
}

/*****************************************************************************/
/*                                                                           */
/*      PATfree(P)                                                           */
/*                                                                           */
/*****************************************************************************/
long PATfree(P)
int     P;
{

  cfree(pattern[P]);
  pattern[P] = NULL;
}

/*****************************************************************************/
/*                                                                           */
/*      PATcommit(P)                                                         */
/*                                                                           */
/*****************************************************************************/
int PATcommit(P)
int     P;
{
  PTNTYPE    pat = pattern[P];
  int        i = 0;

  while (pat->final_states[i] != -1)
     if (pat->final_states[i] == pat->curr_state)
        return OK;
     else
        i++;
  return FAIL;
}
```

```
#define SUBSIZE         20
#define TSIZE           10
#define STSIZE          10
#define SUBOBJSIZE      10
#define PATARGSSIZE     10
#define LOCSIZE         5

#define PRIMARY_PNUM    22

#define CONFLICT        0
#define NOMATCH         1
#define MATCH           2

#define INACTIVE        0

struct inacttype {
        char    object[SSIZE];
        int     index[SUBSIZE];
        int     inact[SUBSIZE];
};

struct ptn_varr {
        char    opname[SSIZE];
        char    object[SSIZE];
        char    membername[PATHSIZE];
};

struct ttype {
        char    opname[SSIZE];
        char    object[SSIZE];
        char    membername[PATHSIZE];
        int     dest_state;       /* destination state for the transition  */
        int     (*loc_action)();
        int     (*action)();      /* point to the action function          */
        struct ptn_varr patargs[PATARGSSIZE];
};

struct stype {                              /* state's trans' data structure */
        int          begin_state;               /* the begin state          */
        struct ttype transitions[TSIZE];        /* the transitions that the */
                                                /*  begin_state matches     */
};

struct subtype {
        char    opname[SSIZE];
        char    object[SSIZE];
        char    membername[PATHSIZE];
        int     subords[SUBSIZE];
};

typedef struct ptntype {                /* pattern's data structure  */
        int     start_state;            /* the start state           */
        int     final_states[STSIZE];   /* a set of final states     */
        int     dead_state;             /* the dead state            */
        int     curr_state;             /* the current stae          */
```

```
        int     update_state;           /* the dest state for update */

        int     (*update_action)();     /* the action for performing */
        struct ptn_varr update_patargs[PATARGSSIZE];

        struct stype ptn_trans[STSIZE]; /* pattern's transition table */
        struct subtype PATsubs[SUBSIZE];
        char    locvar[LOCSIZE][SSIZE];
}P_SP, *PTNTYPE;

extern int  PATinquire();
extern long PATadd();
extern long PATcreate();
extern long PATsubords();
extern int  PATresult();
extern int  PATdecr();
extern long PATfree();
extern int  PATcommit();
```

7 Examples of User-defined Parts and Some Demo Outputs

There are two sample examples to show how the project works. For every example, the machine designers need to take care of the following things:

1. Describe the correctness criteria for every TG and then build the pattern machines according to the specifications of these correctness criteria.

2. Build the TG hierarchy for all the TGs by using the following rules:

   root : must be a TG and each member of the root can either be a CT or another TG.

   node : must be a TG. We can regard the root as a special node where it is at the top of the database.

   leaf : must be a CT.

3. Modify the constant 'PRIMARY_PNUM' which is defined in file 'PAT.h' to be the number of the user-defined pattern machines.

4. Modify the function 'PATcreate()' in file 'PAT.c' to create a proper subordinate pattern for the current example.

There are two new files to be defined by the users in this part. They are:

CCfinder.h : the library file for the database tree of all the TGs
patterns.h : the library file for the pattern machines

Finally we put some demo outputs for each example to show how the program runs.

7.1 Example 1

This is an example for the preparation of a product presentation in another city and we show the TG hierarchy of this in *Figure 7.*

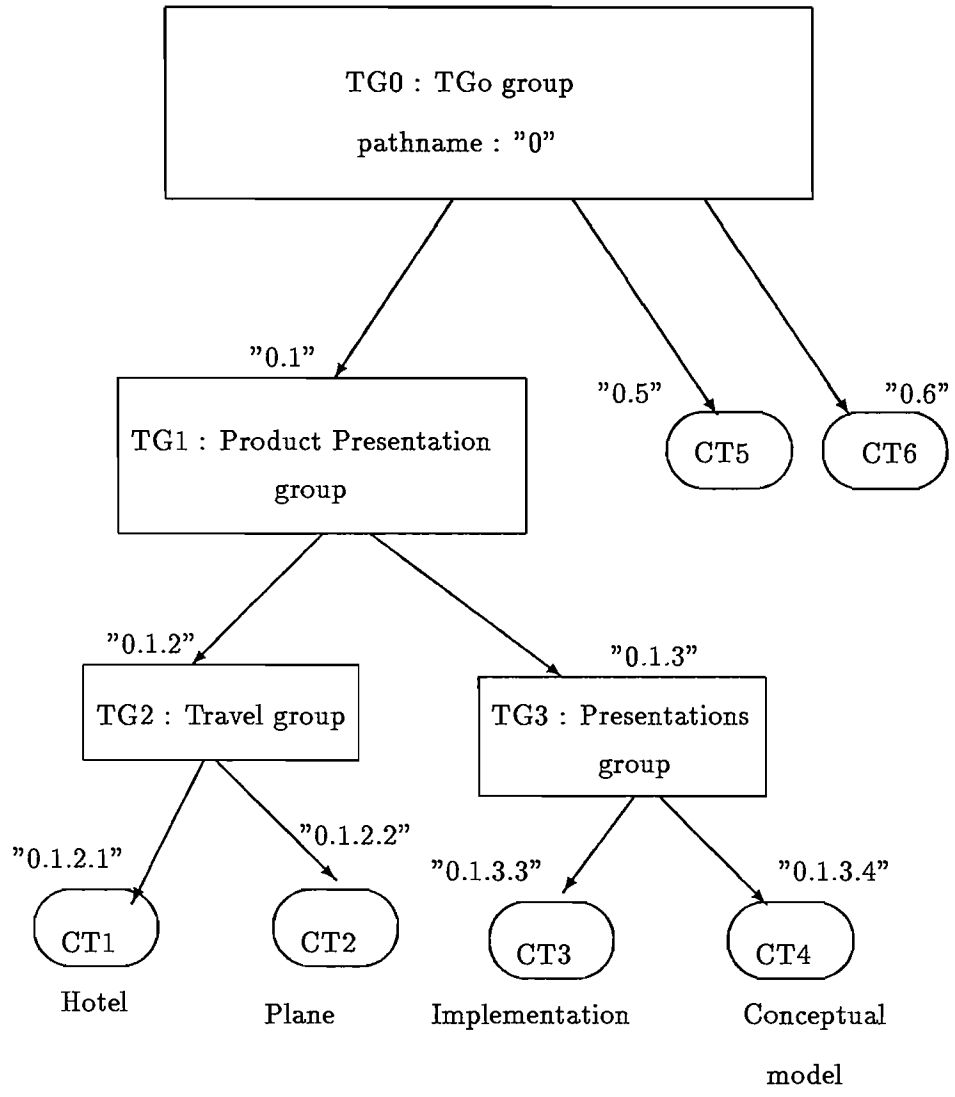


Figure 7: The TG hirearchy for Example 1.

The correctness criteria are as follows.

- TG0 : the system-defined group TGo which is at the top of the TG hierarchy and contains product presentation and two dummy members. There are eleven pattern and conflict criteria in this TG [C0 - C10].

C0 : a time object *t* that can't be modified by a member after another member has read the value.

C1 : object *t* that can't be read or modified by a member after another member has modified it.

C2 : a schedule object *sc* that is compacted after modification.

C3 : object *sc* that can't be modified by a member after another member has read the value.

C4 : object *sc* that can't be read or modified by a member after another member has modified it.

C5 : an itinerary object *itin* that can't be modified by a member after another member has read the value.

C6 : object *itin* that can't be read or modified by a member after another member has modified it.

C7 : a slide object *psl* that can't be modified by a member after another member has read the value.

C8 : object *psl* that can't be read or modified by a member after another member has modified it.

C9 : an account object *pac* that can't be modified by a member after another member has read the value.

C10 : object *pac* that can't be read or withdrawn funds by a member which another member previously deposited or withdraw funds.

- TG1 : the product presentation TG which contains Travel and Presentations two members and seven pattern and conflict criteria [C11 - C17] as follows.

C11 : object *psl* that can't be modified by a member after another has read the value.

C12 : object *psl* that can't be read or modified by a member after another has modified it.

C13 : object *pac* that can't be modified by a member after another member has read the value.

C14 : object *pac* that can't be read or withdrawn funds by a member which another member previously deposited or withdraw funds.

C15 : object *sc* that can't be modified by a member after another member has accessed the value, but suboperations at object *t* can be overwritten.

C16 : an access to object *sc* that is overwritten by a itin operation is reread and reconciled with the current schedule, but overwriting by other types is disallowed.

C17 : object *itin* that can't be modified by a member after another member has accessed the value, and its suboperations are not overwritten.

- TG2 : the Travel TG has Hotel and Plane two CT members and three pattern and conflict criteria as follows.

  C18 : a member whose access to object *itin* is overwritten by another member reads the same object before committing.

  C19 : a member that can't withdraw funds from object *pac* when another member's net deposit is positive.

  C20 : object *pac* that can be read before the object *itin* is modified.

- TG3 : the presentations TG has Implementation and Conceptual Model two CT members and contains a pattern and conflict criterion as follows.

  C21 : object *psl* is printed at least once, and the modified slides are reprinted.

Because the 'PRIMARY_PNUM' and the function 'PATcreate()' given in the previous section are for this example, we don't need to modify them here. So we will list the following files:

CCfinder.h	: TG hierarchy
patterns.h	: pattern machines
demo_data.1a	: a plain example without any tricky service
demo_out.1a	: the demo output of running demo_data.1a

demo_data.1b : an example to show how the local variables work
demo_out.1b : the demo output of running demo_data.1b
demo_data.1c : an example to show how the suboperations work
demo_out.1c : the demo output of running demo_data.1c
demo_data.1d : an example to show how the TG commit works
demo_out.1d : the demo output of running demo_data.1d

```
static FINDER CCfinder[] = {
        {{"0.1", {{0, 1}, {1, 1}, {2, 1}, {3, 1}, {4, 1}, {5, 1},
                  {6, 1}, {7, 1}, {8, 1}, {9, 1}, {10, 1}, -1}},
         {"0.5", {{0, 1}, {1, 1}, {2, 1}, {3, 1}, {4, 1}, {5, 1},
                  {6, 1}, {7, 1}, {8, 1}, {9, 1}, {10, 1}, -1}},
         {"0.6", {{0, 1}, {1, 1}, {2, 1}, {3, 1}, {4, 1}, {5, 1},
                  {6, 1}, {7, 1}, {8, 1}, {9, 1}, {10, 1}, -1}},
         "-1"},

        {{"t", {0, 1, -1}},
         {"sc", {2, 3, 4, -1}},
         {"itin", {5, 6, -1}},
         {"psl", {7, 8, -1}},
         {"pac", {9, 10, -1}},
         "-1"},

        {{"0.1.2", {{11, 1}, {12, 1}, {13, 1}, {14, 1},
                    {15, 1}, {16, 1}, {17, 1}, -1}},
         {"0.1.3", {{11, 1}, {12, 1}, {13, 1}, {14, 1},
                    {15, 1}, {16, 1}, {17, 1}, -1}},
         "-1"},

        {{"psl", {11, 12, -1}},
         {"pac", {13, 14, -1}},
         {"sc", {15, 16, -1}},
         {"itin", {17, -1}},
         {"itin.ar", {16, -1}},
         {"itin.de", {16, -1}},
         "-1"},

        {{"0.1.2.1", {{18, 1}, {19, 1}, {20, 1}, -1}},
         {"0.1.2.2", {{18, 1}, {19, 1}, {20, 1}, -1}},
         "-1"},

        {{"itin", {18, 20, -1}},
         {"pac", {19, 20, -1}},
         "-1"},

        {{"0.1.3.3", {{21, 1}, -1}},
         {"0.1.3.4", {{21, 1}, -1}},
         "-1"},

        {{"psl", {21,-1}},
         "-1"},

        "-2"
};
```

```
extern int Release();
extern int waste();
extern int fct1();
extern int fct2();
extern int fct3();
extern int assign();
extern int add_amt();
extern int delete_amt();

/*      Group TGo machines      */

static struct ptntype pat0[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "t", "0.1", 1, waste, waste}, "-1"}},
         {1,    {{"W", "t", "0.5", 2, waste, waste},
                 {"W", "t", "0.6", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat1[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"W", "t", "0.1", 1, waste, waste}, "-1"}},
         {1,    {{"R", "t", "0.5", 2, waste, waste},
                 {"R", "t", "0.6", 2, waste, waste},
                 {"W", "t", "0.5", 2, waste, waste},
                 {"W", "t", "0.6", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat2[] = {
        0, {0, -1}, -1, 0, 0, 0, {{"-1"}},
        {{0,    {{"W", "sc", "0.1", 1, waste, waste}, "-1"}},
         {1,    {{"Cmp", "sc", "0.1", 0, waste, waste},
                 {"Cmp", "sc", "0.5", 0, waste, waste},
                 {"Cmp", "sc", "0.6", 0, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat3[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "sc", "0.1", 1, waste, waste}, "-1"}},
         {1,    {{"W", "sc", "0.5", 2, waste, waste},
                 {"W", "sc", "0.6", 2, waste, waste},
                 {"Cmp", "sc", "0.5", 2, waste, waste},
                 {"Cmp", "sc", "0.6", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat4[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"W", "sc", "0.1", 1, waste, waste},
                 {"Cmp", "sc", "0.1", 1, waste, waste}, "-1"}},
         {1,    {{"R", "sc", "0.5", 2, waste, waste},
                 {"R", "sc", "0.6", 2, waste, waste},
                 {"W", "sc", "0.5", 2, waste, waste},
                 {"W", "sc", "0.6", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat5[] = {
```

```
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "itin", "0.1", 1, waste, waste}, "-1"}},
         {1,    {{"W", "itin", "0.5", 2, waste, waste},
                 {"W", "itin", "0.6", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat6[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"W", "itin", "0.1", 1, waste, waste}, "-1"}},
         {1,    {{"R", "itin", "0.5", 2, waste, waste},
                 {"R", "itin", "0.6", 2, waste, waste},
                 {"W", "itin", "0.5", 2, waste, waste},
                 {"W", "itin", "0.6", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat7[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "psl", "0.1", 1, waste, waste},
                 {"Pr", "psl", "0.1", 1, waste, waste}, "-1"}},
         {1,    {{"W", "psl", "0.5", 2, waste,waste},
                 {"W", "psl", "0.6", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat8[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"W", "psl", "0.1", 1, waste, waste}, "-1"}},
         {1,    {{"R", "psl", "0.5", 2, waste, waste},
                 {"R", "psl", "0.6", 2, waste, waste},
                 {"W", "psl", "0.5", 2, waste, waste},
                 {"W", "psl", "0.6", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat9[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "pac", "0.1", 1, waste, waste}, "-1"}},
         {1,    {{"Dp", "pac", "0.5", 2, fct1, waste},
                 {"Dp", "pac", "0.6", 2, fct1, waste},
                 {"Wd", "pac", "0.5", 2, fct1, waste},
                 {"Wd", "pac", "0.6", 2, fct1, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat10[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"Wd", "pac", "0.1", 1, fct1, waste},
                 {"Dp", "pac", "0.1", 1, fct1, waste}, "-1"}},
         {1,    {{"Wd", "pac", "0.5", 2, fct1, waste},
                 {"Wd", "pac", "0.6", 2, fct1, waste},
                 {"R", "pac", "0.5", 2, waste, waste},
                 {"R", "pac", "0.6", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

/*      Product Presentation machines   */

static struct ptntype pat11[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
```

```
        {{0,    {{"R", "psl", "0.1.3", 1, waste, waste},
                 {"Pr", "psl", "0.1.3", 1, waste, waste}, "-1"}},
         {1,    {{"W", "psl", "0.1.2", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat12[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"W", "psl", "0.1.3", 1, waste, waste}, "-1"}},
         {1,    {{"R", "psl", "0.1.2", 2, waste, waste},
                 {"W", "psl", "0.1.2", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat13[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "pac", "0.1.2", 1, waste, waste}, "-1"}},
         {1,    {{"Wd", "pac", "0.1.3", 2, fct1, waste},
                 {"Dp", "pac", "0.1.3", 2, fct1, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat14[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"Wd", "pac", "0.1.2", 1, fct1, waste},
                 {"Dp", "pac", "0.1.2", 1, fct1, waste}, "-1"}},
         {1,    {{"Wd", "pac", "0.1.3", 2, fct1, waste},
                 {"R", "pac", "0.1.3", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat15[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "sc", "0.1.3", 1, waste, waste},
                 {"W", "sc", "0.1.3", 1, waste, waste}, "-1"}},
         {1,    {{"W", "sc", "0.1.2", 2, waste, waste},
                 {"Cmp", "sc", "0.1.2", 2, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat16[] = {
        0, {0, 1, -1}, 3, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "sc", "0.1.2", 1, waste, waste},
                 {"R", "sc", "0.1.3", 1, waste, waste},
                 {"W", "sc", "0.1.2", 1, waste, waste},
                 {"W", "sc", "0.1.3", 1, waste, waste}, "-1"}},
         {1,    {{"W", "itin.ar", "0.1.2", 2, waste, waste},
                 {"W", "itin.ar", "0.1.3", 2, waste, waste},
                 {"W", "itin.de", "0.1.2", 2, waste, waste},
                 {"W", "itin.de", "0.1.3", 2, waste, waste}, "-1"}},
         {2,    {{"R", "sc", "0.1.2", 1, waste, waste},
                 {"R", "sc", "0.1.3", 1, waste, waste},
                 {"W", "sc", "0.1.2", 3, waste, waste},
                 {"W", "sc", "0.1.3", 3, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat17[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "itin", "0.1.2", 1, waste, waste},
                 {"W", "itin", "0.1.2", 1, waste, waste}, "-1"}},
```

```
        {1,     {{"W", "itin", "0.1.3", 2, waste, waste}, "-1"}},
        -1}, {{"-1"}}};

/*      Travel machines         */

static struct ptntype pat18[] = {
        0, {0, 1, -1}, -1, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "itin", "0.1.2.1", 1, waste, waste},
                 {"W", "itin", "0.1.2.1", 1, waste, waste}, "-1"}},
         {1,    {{"W", "itin", "0.1.2.2", 2, waste, waste}, "-1"}},
         {2,    {{"R", "itin", "0.1.2.1", 1, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat19[] = {
        0, {0, 1, -1}, 2, 0, 0, 0, {{"-1"}},
        {{0,    {{"Dp", "pac", "0.1.2.1", 1, fct2, assign}, "-1"}},
         {1,    {{"Dp", "pac", "0.1.2.1", 1, fct2, add_amt},
                 {"Wd", "pac", "0.1.2.1", 1, fct2, delete_amt},
                 {"Wd", "pac", "0.1.2.2", 2, fct3, waste}, "-1"}},
         -1}, {{"-1"}}, {"0.0"}};

static struct ptntype pat20[] = {
        0, {0, 1, 2, -1}, 3, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "pac", "0.1.2.1", 1, waste, waste},
                 {"W", "itin", "0.1.2.1", 3, waste, waste}, "-1"}},
         {1,    {{"W", "itin", "0.1.2.1", 2, waste, waste},
                 {"Wd", "pac", "0.1.2.2", 3, fct1, waste}, "-1"}},
         {2,    {{"-1"}}},
         -1}, {{"-1"}}};

/*      Presentations machine   */

static struct ptntype pat21[] = {
        0, {1, -1}, -1, 0, 0, 0, {{"-1"}},
        {{0,    {{"Pr", "psl", "0.1.3.3", 1, waste, waste},
                 {"Pr", "psl", "0.1.3.4", 1, waste, waste}, "-1"}},
         {1,    {{"W", "psl", "0.1.3.3", 2, waste, waste},
                 {"W", "psl", "0.1.3.4", 2, waste, waste}, "-1"}},
         {2,    {{"Pr", "psl", "0.1.3.3", 1, waste, waste},
                 {"Pr", "psl", "0.1.3.4", 1, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

/*****************************************************/
/*                                                   */
/*      functions for local variables and args       */
/*                                                   */
/*****************************************************/
double atof();

int fct1(args, locvar)
char    args[][SSIZE];
char    locvar[][SSIZE];
{

  if (strcmp(args[1], "end"))
```

```
        return OK;
    else
        return ERROR;
}

int fct2(args, locvar)
char    args[][SSIZE];
char    locvar[][SSIZE];
{
    double  amt;

    amt = atof(args[1]);
    if (strcmp(args[1], "end") && amt >= 0.01)
        return OK;
    else
        return ERROR;
}

int fct3(args, locvar)
char    args[][SSIZE];
char    locvar[][SSIZE];
{
    double total;

    total = atof(locvar[0]);
    if (strcmp(args[1], "end") && total >= 0.01)
        return OK;
    else
        return ERROR;
}

int assign(patargs, args, locvar, PATsubs, inactive)
struct ptn_varr    patargs[];
char               args[][SSIZE];
char               locvar[][SSIZE];
struct subtype     PATsubs[];
struct inacttype   inactive[];
{

    strcpy(locvar[0], args[1]);
    return OK;
}

int add_amt(patargs, args, locvar, PATsubs, inactive)
struct ptn_varr    patargs[];
char               args[][SSIZE];
char               locvar[][SSIZE];
struct subtype     PATsubs[];
struct inacttype   inactive[];
{
    double        total;
    double        amt;

    total = atof(locvar[0]);
    amt = atof(args[1]);
```

```
  total += amt;
  sprintf(locvar[0], "%lf", total);
  return OK;
}

int delete_amt(patargs, args, locvar, PATsubs, inactive)
struct ptn_varr    patargs[];
char               args[][SSIZE];
char               locvar[][SSIZE];
struct subtype     PATsubs[];
struct inacttype   inactive[];
{
  double        total;
  double        amt;

  total = atof(locvar[0]);
  amt = atof(args[1]);
  total -= amt;
  sprintf(locvar[0], "%lf", total);
  return OK;
}

/*****************************************************************/
/*                                                               */
/*      build the pattern machines in structure "pattern"        */
/*                                                               */
/*****************************************************************/

static PTNTYPE pattern[PTNSSIZE] = {
  pat0, pat1, pat2, pat3, pat4, pat5, pat6, pat7, pat8, pat9, pat10,
  pat11, pat12, pat13, pat14, pat15, pat16, pat17, pat18, pat19, pat20,
  pat21};
```

```
1       W       psl     0       0.1.3.4
4       0.1     4       str1
        R       psl     0       0.1.3.3
4       0.2     1       1990
8
```

```
The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the W operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result char string.
The new current state of the pattern #8 is state 1.
The new current state of the pattern #12 is state 1.

Return the char string 'str1' back to the invoker '0.1.3.4'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
```

```
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the R operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result integer.
The new current state of the pattern #7 is state 1.
The new current state of the pattern #11 is state 1.

Return the integer '1990' back to the invoker '0.1.3.3'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
```

```
1       Dp      pac     1       100     0.1.2.1
4       0.1     4       fine
        Wd      pac     1       30.5    0.1.2.2
6
1       Wd      pac     1       100     0.1.2.1
4       0.2     1       85
2       Wd      pac     1       30.5    0.1.2.2
7
8
```

```
The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the Dp operation have?
args[1] =
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

lease choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result char string.
The new current state of the pattern #10 is state 1.
The new current state of the pattern #14 is state 1.
The new current state of the pattern #19 is state 1.

Return the char string 'fine' back to the invoker '0.1.2.1'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
```

```
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the Wd operation have?
args[1] =
ct member name =
Pattern #19 makes the operation invocation conflict!

The 'Wd' op invocation makes the patterns conflict!!

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

 he waiting operations are the following:

        opname = Wd
        object = pac
        args[1] = 30.5
        invoker = 0.1.2.2
        ==========================


Finsih listing the queue!

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
```

```
object name =
Besides object, how many arguments does the Wd operation have?
Args[1] =
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result integer.
The new current state of the pattern #19 is state 1.

Return the integer '85' back to the invoker '0.1.2.1'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the Wd operation have?
args[1] =
invoker [a ct member or an object id] =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
```

```
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

CCinv[2] :
        opname = Wd,
        args[] = pac,   30.5,
        invoker = 0.1.2.2,
        opinv = 0.3,
        p_list :
          tg : 0  --  plist :
          tg : 1  --  plist :
          tg : 2  --  plist :

        ==============================


The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
```

```
1       R       pac     0       0.1.2.1
1       0.1     1       300
2       W       itin    0       0.1.2.1
3       W       itin.ar 0       itin    0.2
4       0.2.3   2       33.456
4       0.2     3       100234
8
```

```
The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the R operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result integer.
The new current state of the pattern #9 is state 1.
The new current state of the pattern #13 is state 1.
The new current state of the pattern #20 is state 1.

Return the integer '300' back to the invoker '0.1.2.1'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
```

```
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the W operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the W operation have?
Invoker's object id =
input the invoker's opinv please

create a new subord pattern 22 in CCsubs[0]

create a new subord pattern 23 in CCsubs[1]

create a new subord pattern 24 in CCsubs[2]

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR       ;
```

```
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result float.

Return the float '33.456001' back to the invoker 'itin'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result long.
The new current state of the pattern #6 is state 1.
The new current state of the pattern #17 is state 1.
The new current state of the pattern #18 is state 1.
The new current state of the pattern #20 is state 2.

Return the long '100234' back to the invoker '0.1.2.1'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
```

```
5       3
1       Pr      psl     0       0.1.3.3
        0.1     1       100
5       3
5       1
5       2
5       0
8
```

```
The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
Which TG wants to commit?

TG 3 commit request fails.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the Pr operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;
```

```
Please input the relative type number [0-4] from the menu.
Please input the result integer.
The new current state of the pattern #7 is state 1.
The new current state of the pattern #11 is state 1.
The new current state of the pattern #21 is state 1.

Return the integer '100' back to the invoker '0.1.3.3'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
Which TG wants to commit?

TG 3 has already commited.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
Which TG wants to commit?

TG 1 commit request fails.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
Which TG wants to commit?

TG 2 has already commited.

TG 1 has already commited.
```

```
The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
Which TG wants to commit?

TG 0 has already commited.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
```

7.2 Example 2

This is another example for writing and reading the report with the share rights and the TG hierarchy which is shown in *Figure 8.*

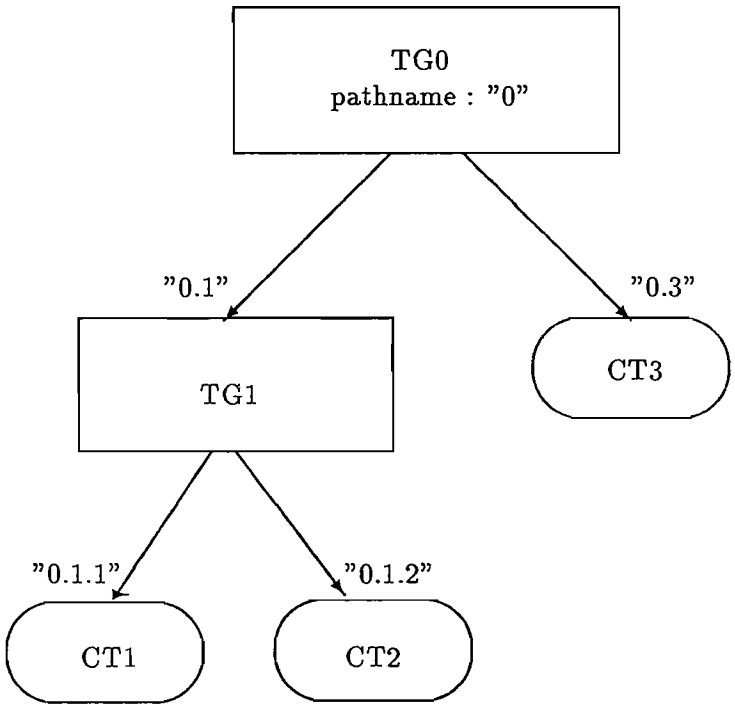


Figure 8: The TG hierarchy for Example 2.

The correctness criteria are as follows.

- TG0 : the system-defined group TGo which is at the top of the TG hierarchy and contains one TG member and one CT member. There are two pattern and conflict criteria in the TG0 [C0 - C1].

  C0 : an object *r.s1* for section 1 in a report that can't be written by a CT member without giving the permission after a TG member

read the report.

C1 : object *r.s1* that can't be rewitten by a TG member without giving the permission after a TG member read the report.

- TG1 : the TG contains two CT members and two pattern and conflict criteria [C2 - C3] as follows.

  C2 : an object *r.s2* for section 2 in a report that can't be written continuously without reading it.

  C3 : an object *r.s3* for section 3 in a report that can't be written continuously without reading it.

The 'PRIMARY_PNUM' in file 'PAT.h' will be changed to 4, and the function 'PATcreate()' in file 'PAT.c' will be modified as in file 'PATcreate' (see the following list). The following files are:

PATcreate : the modified function 'PATcreate()'
CCfinder.h : TG hierarchy
patterns.h : pattern machines
demo_data.2a : an example without using release functions
demo_out.2a : the demo output of running demo_data.2a
demo_data.2b : an example to show how the release functions work
demo_out.2b : the demo output of running demo_data.2b
demo_data.2c : an example to show that ignoring conflict works
demo_out.2c : the demo output of running demo_data.2c

```
/*****************************************************************************/
/*                                                                           */
/*      PATcreate(template, SP, finder, index, count)                        */
/*                                                                           */
/*****************************************************************************/
long PATcreate(template, SP, finder, index, count)
INVTYPE         template;
int             SP;
struct memtype  finder[];
int             index;
int             count;
{
  PTNTYPE       subpat;
  int           i = 0;
  int           j = 0;
  char          member[PATHSIZE];

  subpat = (PTNTYPE) malloc (sizeof(P_SP));
  subpat->start_state = 0;
  subpat->dead_state = 2;
  subpat->curr_state = 1;
  subpat->update_state = 1;
  sprintf(subpat->locvar[0], "%d", count);
  subpat->final_states[0] = 0;
  subpat->final_states[1] = 1;
  subpat->final_states[2] = -1;
  subpat->ptn_trans[1].begin_state = 1;
  while (strcmp(finder[i].name, "-1")) {
     strcpy(member, finder[i++].name);
     if (!strcmp(member, "-2"))
        break;
     else if (!strcmp(member, template->p_list[index].membername)) {
        subpat->ptn_trans[0].begin_state = 0;
        strcpy(subpat->ptn_trans[0].transitions[0].opname, template->opname);
        strcpy(subpat->ptn_trans[0].transitions[0].object, template->args[0]);
        strcpy(subpat->ptn_trans[0].transitions[0].membername, member);
        subpat->ptn_trans[0].transitions[0].dest_state = 1;
        subpat->ptn_trans[0].transitions[0].loc_action = waste;
        subpat->ptn_trans[0].transitions[0].action = waste;
        strcpy(subpat->ptn_trans[0].transitions[1].opname, "-1");
     } else {
        strcpy(subpat->ptn_trans[1].transitions[j].opname, "W");
        strcpy(subpat->ptn_trans[1].transitions[j].object, "r.s1");
        strcpy(subpat->ptn_trans[1].transitions[j].membername, member);
        subpat->ptn_trans[1].transitions[j].dest_state = 2;
        subpat->ptn_trans[1].transitions[j].loc_action = waste;
        subpat->ptn_trans[1].transitions[j++].action = waste;

        strcpy(subpat->ptn_trans[1].transitions[j].opname, "Ap");
        strcpy(subpat->ptn_trans[1].transitions[j].object, "o");
        strcpy(subpat->ptn_trans[1].transitions[j].membername, member);
        subpat->ptn_trans[1].transitions[j].dest_state = 0;
        subpat->ptn_trans[1].transitions[j].loc_action = fct1;
        subpat->ptn_trans[1].transitions[j++].action = remove_sp;
```

```
            strcpy(subpat->ptn_trans[1].transitions[j].opname, "Ap");
            strcpy(subpat->ptn_trans[1].transitions[j].object, "o");
            strcpy(subpat->ptn_trans[1].transitions[j].membername, member);
            subpat->ptn_trans[1].transitions[j].dest_state = 1;
            subpat->ptn_trans[1].transitions[j].loc_action = fct2;
            subpat->ptn_trans[1].transitions[j++].action = decr_sp;
        }
    }
    strcpy(subpat->ptn_trans[1].transitions[j].opname, "-1");
    pattern[SP] = subpat;
}
```

```
static FINDER CCfinder[] = {
        {{"0.1", {{0, 1}, {1, 1}, -1}},
         {"0.3", {{0, 1}, {1, 1}, -1}},
         "-1"},

        {{"r", {0, 1, -1}},
         {"o", {0, 1, -1}},
         "-1"},

        {{"0.1.1", {{2, 1}, {3, 1}, -1}},
         {"0.1.2", {{2, 1}, {3, 1}, -1}},
         "-1"},

        {{"r.s2", {2, -1}},
         {"r.s3", {3, -1}},
         "-1"},

        "-2"
};
```

```
extern int Release();
extern int waste();
extern int fct1();
extern int fct2();
extern int remove_sp();
extern int decr_sp();

/*      Group TG0 machines      */

static struct ptntype pat0[] = {
        0, {0, 1, 2, -1}, -1, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "r", "0.1", 1, waste, waste}, "-1"}},
         {1,    {{"Ap", "o", "0.3", 2, waste, Release,
                                {{"R", "r", "0.1"}}}, "-1"}},
         {2,    {{"R", "r", "0.1", 1, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat1[] = {
        0, {0, 1, 2, -1}, -1, 0, 0, 0, {{"-1"}},
        {{0,    {{"R", "r", "0.3", 1, waste, waste}, "-1"}},
         {1,    {{"Ap", "o", "0.1", 2, waste, Release,
                                {{"R", "r", "0.3"}}}, "-1"}},
         {2,    {{"R", "r", "0.3", 1, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

/*      Group TG1 machines      */

static struct ptntype pat2[] = {
        0, {0, 1, 2, -1}, 3, 0, 0, 0, {{"-1"}},
        {{0,    {{"W", "r.s2", "0.1.1", 1, waste, waste},
                 {"W", "r.s2", "0.1.2", 1, waste, waste}, "-1"}},
         {1,    {{"R", "r.s2", "0.1.1", 2, waste, waste},
                 {"R", "r.s2", "0.1.2", 2, waste, waste},
                 {"W", "r.s2", "0.1.1", 3, waste, waste},
                 {"W", "r.s2", "0.1.2", 3, waste, waste}, "-1"}},
         {2,    {{"W", "r.s2", "0.1.1", 1, waste, waste},
                 {"W", "r.s2", "0.1.2", 1, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

static struct ptntype pat3[] = {
        0, {0, 1, 2, -1}, 3, 0, 0, 0, {{"-1"}},
        {{0,    {{"W", "r.s3", "0.1.1", 1, waste, waste},
                 {"W", "r.s3", "0.1.2", 1, waste, waste}, "-1"}},
         {1,    {{"R", "r.s3", "0.1.1", 2, waste, waste},
                 {"R", "r.s3", "0.1.2", 2, waste, waste},
                 {"W", "r.s3", "0.1.1", 3, waste, waste},
                 {"W", "r.s3", "0.1.2", 3, waste, waste}, "-1"}},
         {2,    {{"W", "r.s3", "0.1.1", 1, waste, waste},
                 {"W", "r.s3", "0.1.2", 1, waste, waste}, "-1"}},
         -1}, {{"-1"}}};

/*********************************************/
/*                                           */
/*      local functions and actions          */
/*                                           */
```

```
/*************************************************/

int fct1(args, locvar)
char    args[][SSIZE];
char    locvar[][SSIZE];
{
  if (!strcmp(locvar[0], "1"))
        return OK;
  else
        return ERROR;
}

int fct2(args, locvar)
char    args[][SSIZE];
char    locvar[][SSIZE];
{
  if (atoi(locvar[0]) > 1)
        return OK;
  else
        return ERROR;
}

int remove_sp(patargs, args, locvar, PATsubs, inactive)
struct ptn_varr    patargs[];
char               args[][SSIZE];
char               locvar[][SSIZE];
struct subtype     PATsubs[];
struct inacttype   inactive[];
{
  strcpy(locvar[0], "0");
  return OK;
}

int decr_sp(patargs, args, locvar, PATsubs, inactive)
struct ptn_varr    patargs[];
char               args[][SSIZE];
char               locvar[][SSIZE];
struct subtype     PATsubs[];
struct inacttype   inactive[];
{
  int   i;

  i = atoi(locvar[0]) - 1;
  sprintf(locvar[0], "%d", i);
  return OK;
}

/*****************************************************************/
/*                                                               */
/*      build the pattern machines in structure "pattern"        */
/*                                                               */
/*****************************************************************/

static PTNTYPE pattern[PTNSSIZE] = {pat0, pat1, pat2, pat3};
```

```
1       R       r       0       0.1.2
4       0.1     4       string
1       W       r       0       0.3
4       0.2     1       100
8
```

```
The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the R operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result char string.
The new current state of the pattern #0 is state 1.

Return the char string 'string' back to the invoker '0.1.2'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
```

```
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the W operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result integer.

Return the integer '100' back to the invoker '0.3'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
```

```
1       R       r       0       0.1.2
3       R       r.s1    0       r       0.1
4       0.1.2   1       100
4       0.1     4       string
1       W       r.s1    0       0.3
6
1       Ap      o       0       0.3
4       0.3     2       222.44
1       W       r       0       0.3
3       W       r.s1    0       r       0.4
7
8
```

```
The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the R operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the R operation have?
invoker's object id =
input the invoker's opinv please

create a new subord pattern 6 in CCsubs[0]

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
```

```
opinv =
Possible types of the op result:
        (0) ERROR       ;
        (1) integer     ;
        (2) float       ;
        (3) long        ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result integer.

Return the integer '100' back to the invoker 'r'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR       ;
        (1) integer     ;
        (2) float       ;
        (3) long        ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result char string.
The new current state of the pattern #0 is state 1.

Return the char string 'string' back to the invoker '0.1.2'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
```

```
object name =
Besides object, how many arguments does the W operation have?
ct member name =
Pattern #6 makes the operation invocation conflict!

The 'W' op invocation makes the patterns conflict!!

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

The waiting operations are the following:

        opname = W
        object = r.s1
        invoker = 0.3
        ========================


Finsih listing the queue!

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the Ap operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
```

```
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result float.
The new current state of the pattern #0 is state 2.

Return the float '222.440002' back to the invoker '0.3'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the W operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
```

```
Besides object, how many arguments does the W operation have?
invoker's object id =
input the invoker's opinv please

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

CCinv[2] :
        opname = W,
        args[] = r,
        invoker = 0.3,
        opinv = 0.4,
        p_list :
          tg : 0  --  plist :

        ================================

        opname = W,
        args[] = r.s1,
        invoker = r,
        opinv = 0.4.5,
        p_list :
          tg : 0  --  plist :

        ================================


The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
```

```
1       R       r       0       0.1.2
3       R       r.s1    0       r       0.1
4       0.1.2   1       100
4       0.1     4       char_str
1       W       r.s1    0       0.3
6
1       W       r       0       0.3
3       W       r.s1    0       r       0.3
7
8
```

```
The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the R operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the R operation have?
invoker's object id =
input the invoker's opinv please

create a new subord pattern 6 in CCsubs[0]

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
```

```
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result integer.

Return the integer '100' back to the invoker 'r'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Inputing the op result information:
opinv =
Possible types of the op result:
        (0) ERROR      ;
        (1) integer    ;
        (2) float      ;
        (3) long       ;
        (4) char string;

Please input the relative type number [0-4] from the menu.
Please input the result char string.
The new current state of the pattern #0 is state 1.

Return the char string 'char_str' back to the invoker '0.1.2'.

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
```

```
Besides object, how many arguments does the W operation have?
ct member name =
Pattern #6 makes the operation invocation conflict!

The 'W' op invocation makes the patterns conflict!!

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

The waiting operations are the following:

        opname = W
        object = r.s1
        invoker = 0.3
        ========================


Finsih listing the queue!

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the W operation have?
ct member name =

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
```

```
        (8) quit.

Please choose an option number from above [1-8].

Input the op invocation information:

operation name =
object name =
Besides object, how many arguments does the W operation have?
invoker's object id =
input the invoker's opinv please

The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].

CCinv[1] :
        opname = W,
        args[] = r,
        invoker = 0.3,
        opinv = 0.3,
        p_list :
          tg : 0  --  plist :

        ================================

        opname = W,
        args[] = r.s1,
        invoker = r,
        opinv = 0.3.4,
        p_list :
          tg : 0  --  plist :

        ================================


The forms of interaction:
        (1) op1 invocation from ct;
        (2) op invocation from queue;
        (3) op2 invocation from op1;
        (4) op result;
        (5) TG commit request;
        (6) list queue;
        (7) list CCinv in CC module;
        (8) quit.

Please choose an option number from above [1-8].
```