KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology

Control Structures for Parallel Computing in NUT

Mattin Addibpour

Department of Teleinformatics

KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology

Control Structures for Parallel Computing in NUT

Mattin Addibpour

MSc thesis

May 1995

This work has been funded by the Swedish National Board for Technical and Industrial Development (NUTEK) under grant number 9303405-2.

TRITA–IT R 95:18
ISSN 1103–534X
ISRN KTH/IT/R—95/18—SE

Department of Teleinformatics

Abstract

Specification language of NUT has been recently extended with functions for distributed computing. This extension can be used for developing a collection of control structures for parallel compositional programming. The purpose of this thesis is to support distributed and coarse grained parallel computing in the NUT system, on the basis of the rNUT distributed programming toolkit.

This work consists of two parts. In the first part an idea of an environment for domain-specific parallel programming is proposed. The proposed environment consists of a library of control structures, designed as classes in NUT, using the rNUT toolkit. A set of experimental control structures and their related applications have been implemented.

In the second part an approach to a graphical language for parallel programming in NUT has been proposed. The proposed language simplifies parallel programming in NUT by using the expressiveness of the graphical language. An experimental implementation of the graphical parallel language has been done using NUT Scheme Editor.

Acknowledgments

I would like to thank Prof. Enn Tyugu for his strong encouragement and continuous guidance during all stages of this work. I also wish to thank my supervisor Vladimir Vlasov for his support. Special thanks to Tarmo Uustalu for his support and encouragement and Diana Sidarkeviciute for introducing the SoftSpy architecture and her cooperation.

This work is an experimental part of the project “Knowledge-Based Software Tools” supported by the Swedish National Board for Technical and Industrial Development (NUTEK) under grant number 9303405-2.

Contents

1.0 Introduction

1.1 Purposes and Goals

The purpose of this thesis is to support distributed and coarse grained parallel computing in the NUT system, on the basis of the rNUT distributed programming toolkit.

The goals are described as follows:

- experiments will be done on a set of control structures and their related applications. These control structures could be designed in the form of classes in NUT. Each control structure should have a predefined topology and should be related to a specific class of applications.

  The outcome of this part is a software package supporting control structures for parallel computing in NUT.

- an approach to a graphical parallel language in NUT should be proposed and tested by using NUT Scheme Editor. This graphical language should reduce the effort of parallel programming in NUT by using the expressiveness of the graphical language.

  The outcome of this part is a software package supporting graphical parallel programming in NUT.

1.2 Environment Notes

1.2.1 Terminology

- **EDA - The Extended Dataflow Actor model.**
- **global object - persistent objects at the workspace level (top level).**
- **GPLN - Graphical Parallel Language in NUT.**
- **NUT - An object-oriented/declarative programming environment.**
- **rNUT - The distributed programming toolkit for NUT.**
- **PAR-class - A control structure implemented as a class in NUT.**
- **private object - global objects which can be only accessed by the local process.**
- **public object - global objects which can be accessed by remote processes.**
- **PVM - Parallel Virtual Machine.**

1.2.2 NUT

The NUT system is a knowledge-based programming environment which supports development of large programs in the object-oriented style. It is compatible with C language, runs under UNIX, and uses the X Windows System.

The NUT programming language is based on two paradigms, (procedural and modular) object-orientation and (declarative) structural synthesis of programs. It combines object-oriented programming (with multiple inheritance, parametric polymorphism, and a flexible mechanism for message passing), visual programming, and automatic program construction paradigms. Besides the ordinary usage, classes in NUT serve as specifications which can be used for constructing programs automatically [Tyu94a].

The knowledge base of the NUT system is divided into packages. A package is the largest modular unit in NUT. A package constitutes an environment where a process described by a script in the workspace can run. It is a collection of programs, specifications and data, related to a particular problem domain. More precisely, it is a collection of classes, global (persistent) objects and graphic views (icons, images and schemes) intended for usage in the problem domain. A package can be visualized in a NUT main window, Figure 1. It shows a list of classes (Point, baseset,...), a list of global objects (q, t, w), and a script in a workspace for immediate execution.

NUT has a scheme editor for specifying classes graphically, by drawing their schemes. This can be done, whenever problems are of such a nature that they can be specified in terms of classes developed for a particular problem domain in advance. The way from a graphical specification to expected results is the following:

graphical schema -> textual specification -> constraint network ->
synthesized algorithm -> results of computation.

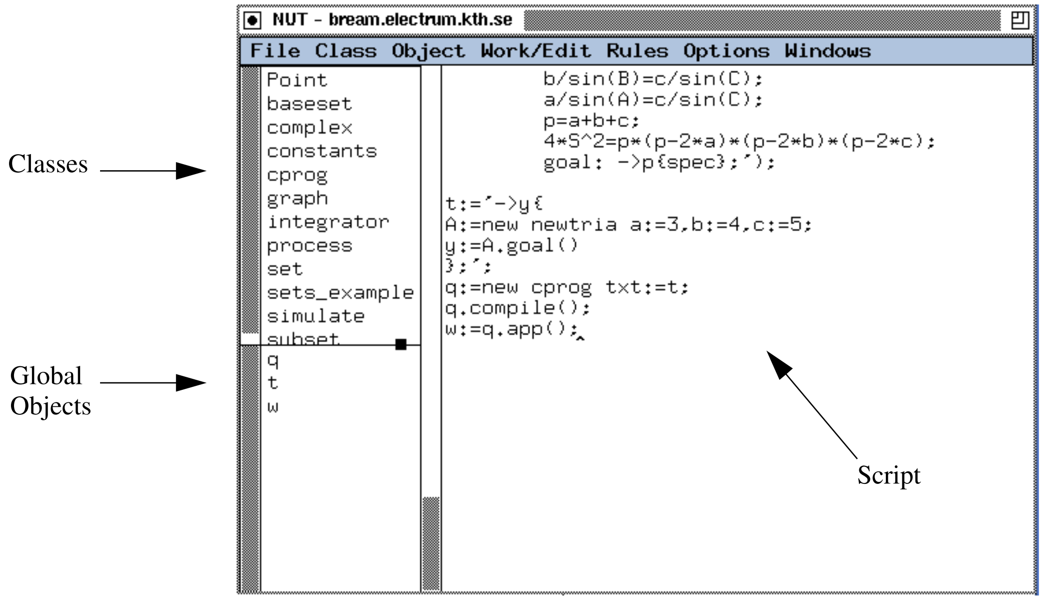


FIGURE 1. THE NUT WINDOW

In many cases, the intermediate steps are completely hidden from a user. Using the Scheme Editor, a user can develop its own graphical language and use it for specifying programs in his problem domain.

The NUT system is self-contained and large programs can be written completely in NUT. However, NUT is open to extensions written in C, and existing C libraries can be used from NUT (and managed by NUT, if proper classes are defined). It is also possible to call NUT from C and, for example, use it as an interactive graphical front-end for large C programs [Kop94b]. These are rather new features of NUT, which have been exploited in this work.

More detailed description of the NUT system can be found in [Tyu94a, UKK+94].

1.2.3 rNUT

- **Introduction**

The rNUT toolkit is an extension of the NUT system version 2.7 developed at KTH [VTA94a]. It consists of a library librnut and a daemon nutd. The library librnut contains low level NUT routines which provide support for coarse-grained (MIMD) distributed programming in NUT. These functions use the PVM distributed computing platform for parallel computation at the level of Unix processes, using message passing for communication. The PVM provides a virtual multiprocessor composed of heterogeneous computers in a network. To support interprocess communication, librnut contains routines for exchanging classes, scripts and objects between NUT processes. The synchronization and communication mechanism of rNUT is based on the Extended Dataflow Actor model, EDA, developed at KTH [VTA94b].

- **NUT Processes on PVM**

A distributed program developed by means of the rNUT toolkit represents a collection of dynamically created communicating processes running on PVM in a network. Each process runs in a separate NUT environment consisting of a NUT package. Each process can have an open NUT window where the lists of classes and objects are visible together with the script. The environment of a process can be changed dynamically by the process itself or by other processes.

The process that is started first is called a root process. Any process can spawn a number of new processes using one function call. These processes together constitute a collection of processes called a grid. Figure 2 shows a snapshot of a dynamically changing pool of NUT processes running on PVM in a network. Processes in one and the same grid can be identified by their indices in the grid, as well as by their task identifiers (tid). Processes in different grids can be addressed only by their task identifiers which are unique in the running rNUT.

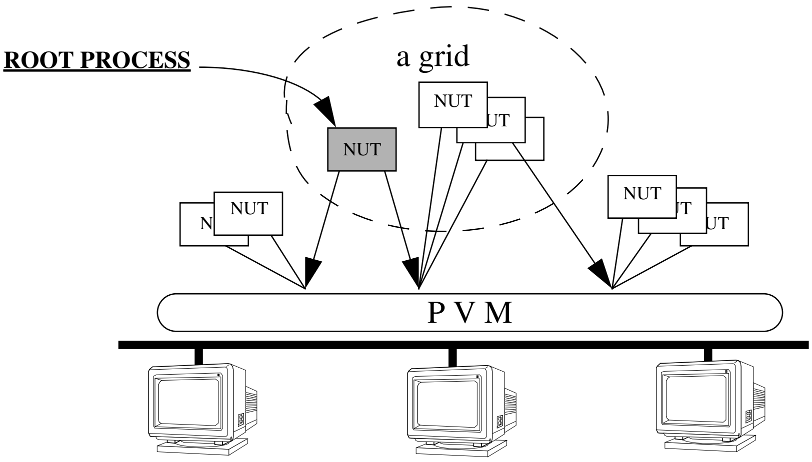


FIGURE 2. A Tree of NUT Processes

- **Communication operations**

In the rNUT, functions are provided for the following kinds of operations:

- synchronization and object passing (EDA communication operations),
- passing classes,
- passing scripts,
- remote process control.

The first three operations are briefly described in the following, a more detailed description can be found in [VTA94a]. The parallel virtual machine (PVM) is not discussed in this report, but detailed description can be found in [G+94, SGD+94].

Passing of Objects:

The rNUT communication model, for passing objects between NUT processes, is based on the communication aspects of the Extended Dataflow Actor model, EDA. A formal description and software implementation of the EDA model can be found in [Tho94, VTA94b].

An EDA object contains local and shared variables and a thread of control. All shared variables form a space of shared memory which is accessible from each object. In EDA, a data passing between local memory of objects is presented by set of special store and fetch operations on a spaces of local and shared variables. Defining shared memory operations, EDA recognizes three kinds of shared variables: *x, i* and *s*, each with special synchronization requirements [Tho94].
Semantics of typed shared variable of EDA are used in rNUT on the level of communication model and has an extension in terms of operations to simplify rNUT application design and make it more convenient and flexible for NUT programming technology (see table 1).

The main difference between rNUT and EDA model is the lack of actual shared memory in the rNUT model. In the rNUT model the shared objects are distributed among the NUT processes. Each shared object can be accessed by using the name of the object and the tid of the NUT process which contains the shared object. When a NUT process wants to pass an object to another NUT, it must store the object in an intermediate object. Due to lack of shared memory space in NUT, the shared object must be stored in the Object Memory of a NUT process (as a global object). Other processes can now fetch the object accessing this shared object. In this report, we distinguish between EDA and rNUT model by introducing the following terminology. In order to make a difference between global objects used only by local processes and those used by other remote processes, we call the latter for ***public objects***. Others objects (and even components of these objects) which are used only by the local process are termed as ***private objects***.
Note that the difference between ***private*** and ***public objects*** are not supported at the lower level and if it is needed, it must be supported by the programmer at the NUT level.

Name	Action	Full	Empty
x-fetch	extract a meaning value from a ***public object*** to ***private***	extract	suspend
i-fetch	copy a meaning value from a ***public object*** to ***private***	copy	suspend
s-fetch	extract any value from a ***public object*** to ***private***	extract	return *empty*
u-fetch	copy any value from a ***public object*** to ***private***	copy	return *empty*
x-store	store a ***private*** value to an empty ***public object***	suspend	store
i-store	store a ***private*** value to an empty ***public object***	skip	store
s-store	store a ***private*** value to an empty ***public object***	buffering	store
u-store	update a ***private*** value to ***public object***	update	store

Table 1: Communication Operations

In other words, object passing is performed in the form of fetch/store operations between ***private*** and ***public objects***, using communication operation types, described at the table 1. These operation can be combined in order to achieve the required communication model. Figure 3 shows a Combination of a non-blocking store with a blocking extract operation. Note that A and C are ***private objects*** belong to different processes. B is a ***public object***, and can belong to any of the NUT processes.

$$A \xrightarrow{\text{sstore}} B$$

$$C \xleftarrow{\text{xfetch}} B$$

FIGURE 3. Combination of a non-blocking store with a blocking extract operation.

Passing of classes

It is possible to pass the classes between remote NUT processes. rNUT also copies the superclass if needed. The passing classes will be compiled at the run time. When an object is passed to a NUT process which does not contain the class of the object, the class will be automatically copied and compiled.

Passing of script

It is also possible to exchange the script of NUT processes. The script passing can be followed by a request to perform the script.

- **Message passing:**

By combining object, class and script passing, it is possible to create different environments for distributed application.

Messages in rNUT can be objects or scripts[1], the difference depends on which application is concerned.

Two combinations have been exploited in our approach:

- Using ***public objects*** as the values passed between processes and also as textual messages, for requesting actions to be performed. The script is only used to initialize an object and to start execution in the remote process.
- Using scripts as messages between processes, this has been used in the SoftSpy. In this approach the NUT process can be seen as a group of objects which uses object and script passing in order to perform the requested actions.

There are other kinds of message sending in the form of appending or modifying the script of a NUT process. It is also possible to change the whole package in a process, which is useful for process initialization.

1. Passing classes changes the state of the class library in the NUT package. Here we have not considered this as massage passing between NUT processes.

1.3 Related Works

We have found the following four works essentially related to our approach:

GRIDS. GRIDS is a parallel programming system for grid-based algorithms. A GRIDS program consists of two parts, a description of the computation topology (the grid) and a description of the algorithm to be executed on the element of the topology. The basic idea of the GRIDS programming model is the separation of the problem description in form of an underlying topology from the specification of solution algorithms. The goal is to offer access to efficient parallel computing without the need for explicit parallel programming [G+93].

PL/LL language. The PL/LL programming model is based on two language levels, the PL (programming language) level for defining the object classes required for an application, and the LL (linking language) level for defining a configuration of objects constituting a particular application program. The LL allows hierarchic composition of modules. An application of LL is for describing parallel programs where modules are distributed over the nodes of a multiprocessor system [TL95]. The authors propose that the graphical indication of module instantiation and connections will simplify the programming in LL. We have used this idea in order to implement a experimental graphical language for parallel programming in NUT. The operational level in our approach serves the purpose of PL, and the structural level is analogous to LL. We mention here that the test example in PARDIF is also inspired by the Dirichlet example described in [VTA94b].

HeNCE. The heterogeneous network computing environment (HeNCE) is an integrated graphical environment for creating and running parallel program over a heterogeneous collection of computers. It is built on top of the PVM. The HeNCE philosophy of parallel programming is to allow a programmer to specify the parallelism of a computation graphically. Key to HeNCE is a graphical language based on directed graphs that describe the parallelism and data dependencies of an application. Nodes in the graphs represent conventional Fortran or C subroutines and the arcs represent data and control flow. Scalars, vectors and two-dimensional arrays are supported for parameter passing in HeNCE [B+94].

Structural Blanks. structural blanks is a method for expressing computations based on recurrence relations. This method relies on a separation between 1) the structural part describing the data dependency pattern of the recurrence, 2) the definition of the expression to be computed as the computational aspect and 3) the initialization and definition of outputs [CH94].

In [Cyr88] the author proposes an approach to loop program synthesis. The main concepts in his approach are:

F-modules, which perform computation according to recurrence relations on arrays;

S-modules, is a library of array traversing algorithms;

and finally, structural linkages between S- and F-modules.

Note that the last work expresses methods for computations based on recurrence relations.

2.0 Control Structure For Parallel Computing

The flexibility of rNUT high level communication functions combined with the open programming environment NUT can be exploited in order to make different distributed programming environments. The intention is to construct and run parallel branches of computations automatically, using a set of predefined control structures. These control structures are implemented in the form of classes in NUT. Once preprogrammed, the control structures are expected to constitute a domain-specific parallel programming toolkit.

2.1 General Idea

Here we are using an idea presented in [VTA94a]:

The NUT specification language enables one to describe computations in a compositional way, by combining computations described in subproblems of relations into larger programs [UKK+94]. A relation with subproblems can be considered as a control structure which composes a new algorithm from existing algorithms for solving subproblems. By using rNUT functions, relations with independent subproblems can be programmed in such a way that they will specify parallel execution, i. e. they will compose algorithms of parallel computations. This can be visualized graphically, as shown in Figure 4.

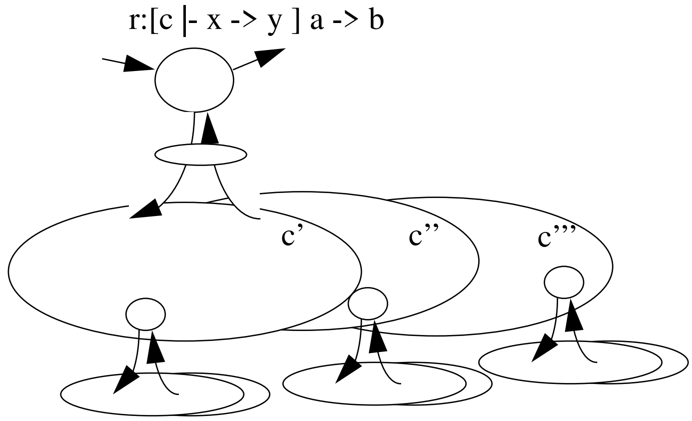


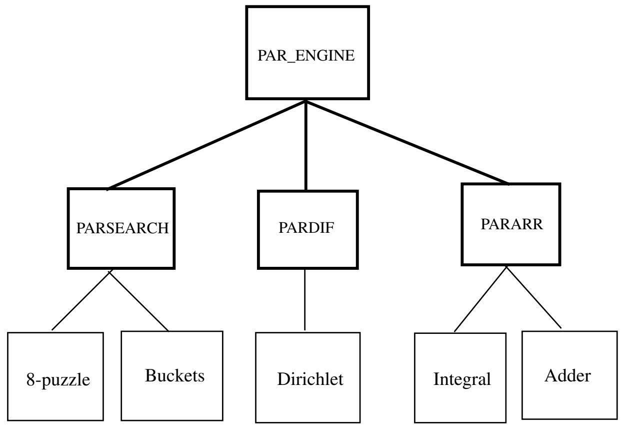
FIGURE 4. Relation r with subproblems controls computations on copies of a class

The relation **r** in Figure 4 creates dynamically a number of environments (c', c'', c''') each of which is a copy of the class c given in the subproblem specification. Using functions for spawning processes, instead of a subproblem call, one can run computations on classes in parallel. As the Figure 4 illustrates, the parallel computing control structures in NUT can be used in a nested way. Development of various control structures in the form of relations with subproblems will be a task of an application software developer. Once preprogrammed, the control structures are expected to constitute a domain-specific parallel programming toolkit.

Example of such structures are discussed in [Kop94a]. The author has used the PVM directly for programming operators like **pararr** for parallel processing of elements of arrays.

An Experimental Environment for Parallel Computing:

Figure 5 shows the idea of an environment for domain-specific parallel programming.



<- Administrative Control Structure

<- Problem-oriented Control Structures

<- Application level

FIGURE 5. An Environment for Domain-Specific Parallel Programming

The task of developing parallel programs consists of a number of sub-tasks: parallel algorithm design, problem partitioning, mapping, communication and synchronization management and resource management. A domain-specific parallel programming environment in the form of a hierarchical set of NUT classes has been realised to facilitate design and implementation of parallel applications in NUT, and to achieve their efficient processing on PVM. A set of control structures supports automatic mapping, communication and synchronization management as well as resource management in the development of parallel applications. Control structures are divided into two levels: administrative and problem-oriented. These control structures are then used by classes at the third (application) level, in order to construct a parallel application.

At the top of the class inheritance hierarchy we see class PAR_ENGINE, which manages the configuration of PVM, spawns and administrates the processes needed by the underlying classes.

On the second level, we have the problem-oriented control structures. These classes are implemented to be used in a domain-specific kind of problems. They hide all aspects of process creation, communication and synchronisation from the application level. For example, class PARSEARCH is implemented for solving the breadth-first search problems, PARDIF is implemented for supporting grid-based computation and PARARR is implemented for parallel processing of elements in an array.

Finally, the application level contains classes which formulate the specific problem in a sequential manner. In the case of PARDIF, Dirichlet is a class which formulates the Dirichlet problem [CT92], Adder adds arrays in parallel and the 8-puzzle class solves a simplified version of 15-puzzle problem.

The intention of the presented work in this chapter, is to construct parallel branches of computations automatically, used in the experimental environment described at the above.

2.2 PAR_ENGINE

2.2.1 General Idea

The rNUT functions are build on the top of PVM. They are still primitive functions at the NUT level. It is desirable to have a higher level process control and administrator, which can start the PVM, add hosts, create and kill the processes.

In order to support the other control structures with higher order process control functions, we have collected a set of process control routines in a class (PAR_ENGINE).

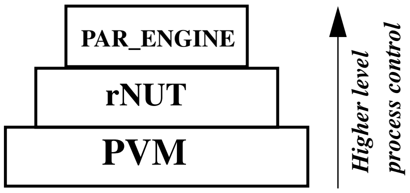


FIGURE 6. Process control hierarchy.

Once an instance of PAR_ENGINE is created, it manages a process pool under a NUT session. This global object, called **pool_manager**, can be used by the other PAR-classes for creating a process pool, using already spawned processes or increasing number of processes in the pool.

It should exist only one instance of PAR_ENGINE in each NUT process. As a result, different PAR-classes can use the same **pool_manager** in a NUT process.

Because of NUT's starting overhead[1], it seems to be more rational to keep the spawned processes alive during a NUT session instead of killing them. However, this is only considered for the root process. If PAR_ENGINE starts in a child process, the spawned pool will be exited after performing the task.

Another observation is that performance decreases significantly when a remote process is started in the root host. Therefore we avoid spawning remote processes in the root host. This is unfortunately not supported by rNUT functions, and the choice is to spawn and see if the process is started at the root host, then the process is killed and a new process is spawned until success. This means that we spawn n grids each containing one process instead of one grid containing n process. This problem should be considered in the future development of rNUT.

pool_manager can be created by making an instance of PAR_ENGINE in the workspace. Once created, **pool_manager** is responsible for pool administration. By calling methods in object window of **pool_manager** or in the script, one can change the number of processes in the pool, kill the pool, change the mode of processes to open or close, or add hosts to the PVM.

1.The NUT starting overhead is not a major factor when the computation time is large compare with communication time, overhead becomes too short to be considered.

The PAR-classes should check if an **pool_manager** is already created, otherwise they should create a **pool_manager** and should store it as a global object using *u-store* operation. Note that *u* operation can be used internal by the process for updating or copying a global object. When a PAR-class uses **pool_manager**, it may change the state of the pool, the state should be updated when PAR-class has performed the task and released the pool (by using *u-store* operation).

The PAR_ENGINE can be used as a superclass in other PAR-classes. Here we choose to use **pool_manager** as a global pool administrator in the form of a global object; consequently, **pool_manager** can be reused by other classes.

2.2.2 Design

Following methods are used in the PAR_ENGINE:

- **ADDHOSTS**:

  This method uses rnut_addhost routine in order to change the configuration of PVM. This method uses a default array of host names for initialisation of the PVM configuration.

- **MAKEPOOL**:

  This method makes a pool of m processes. It returns the tids of the processes as an array of num. If a pool with n processes already exists, it will extend the pool by spawning m-n new processes. If m<n a pool of m processes returns. The window mode of NUT window in the spawned processes is depend on the window mode of the root process. As default the same package will be loaded into the spawned processes. The **pool_manager** checks the window mode of the root process and spawns new processes with the same window mode.

  This method avoids to spawn new processes in the root host (for the reason we discussed in the 2.2.1), child processes use the method FINDROOT in order to localize the root host.

- **KILLPOOL**:

  Kills the process pool, previously spawned by the MAKEPOOL.

- **OPEN**:

  Changes the mode of the NUT window into open window interface.

- **CLOSE**:

  Changes the mode of the NUT window into close window interface.

- **FINDROOT**:

  This method locates the root host. This is specially used by the child processes to locate the root host and to avoid spawning a remote process in the root host.

Default Values.

The following default values are used in the PAR_ENGINE:

mode := 2; % open window interaction mode
addhost := [‘sole’, ‘dolphin’,,,,,,,,,,,,,,,,,,]; %default hosts
root_host := FINDROOT();   %find the root host
pack := rnut_mypack();%load the same package into remote processes

2.2.3 Example

In Figure 7, we see the **pool_manager** and a process pool with two remote processes, spawned in two different hosts, with open window mode.

The process control can be managed by calling the methods, directly from the object window of the **pool_manager**. The **pool_manager** is created from the workspace of NUT, but it could be also created and used from any object of PAR- classes.

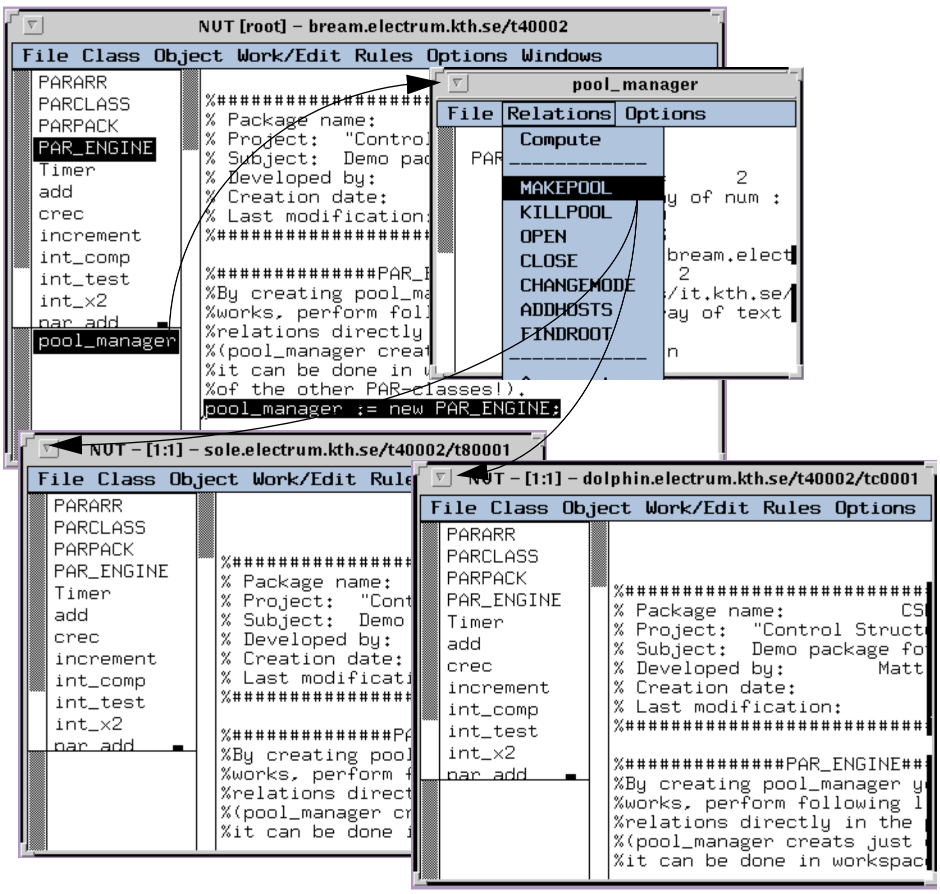


FIGURE 7. Administrating a process pool by using PAR_ENGINE.

2.3 ParSearch

2.3.1 General Idea

Here we apply a control structure (ParSearch) for solving a search problem by means of breadth-first search, BFS. ParSearch can be used as a superclass in the application-oriented class UsrParSearch.

2.3.2 Search Method

There exists various different search methods depending on the nature of search problems. We choose an exhaustive search method, more specific a parallel version of a breath-first search.

- **Breadth-first search**

The breadth-first search can not be parallelized effectively. The reason is that in BFS we are mostly interested in the shortest path way. To find this, each level must be searched before starting next level. Another limitation is that we should avoid to proceed the nodes which have already been checked (see Figure 8).

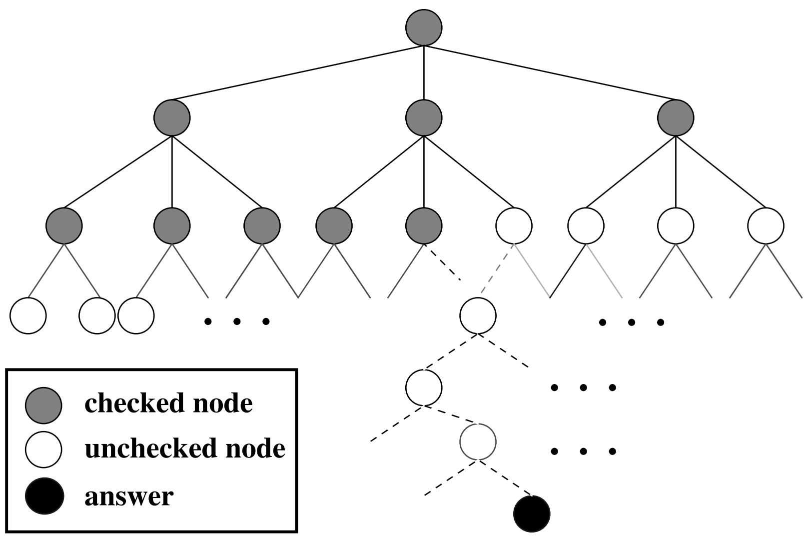


FIGURE 8. Breadth-first search

2.3.3 Process Pool

The algorithm implemented here has a root process and a process pool containing NUT processes for exploring open nodes of the search tree. We search in one level and at the same time we produce children in the next level. We don't start searching in the next level before we finish the previous level (see Figure 9).

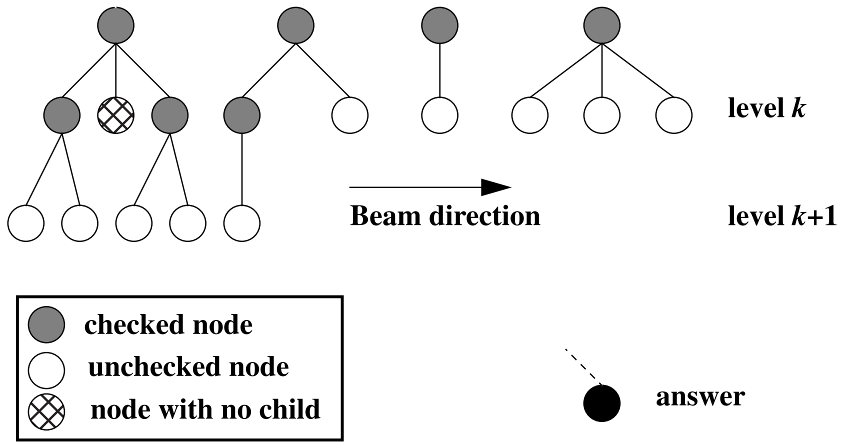


FIGURE 9. Beam search: the first beam is searching in level k, the second beam produces nodes in the next level (k+1).

Root process has two queues (**queue1** & **queue2**), they contain a number of jobs (each job has all the information about a tree node to be examined). **Queue1** contains all the jobs related to nodes of the actual searching level (level k in Figure 9). Root process gets a job from **queue1** finds a free process in the pool and sends the job to that process (see Figure 10). After checking the job, the process will send one of the following three messages to root:

1. **more_childs:** means that the answer is not found in this node, process produces more children after modifying the state of node. Root gets the new states (belonging to the next level) and puts them in **queue2** and notes that a process is free.
2. **no_more_child:** means that the answer is not found in this node and the node is actually a leaf in the tree. Root note that a process is free.
3. **answer:** the answer is found and has the shortest path. Root sends exit message to all processes and notifies the answer.

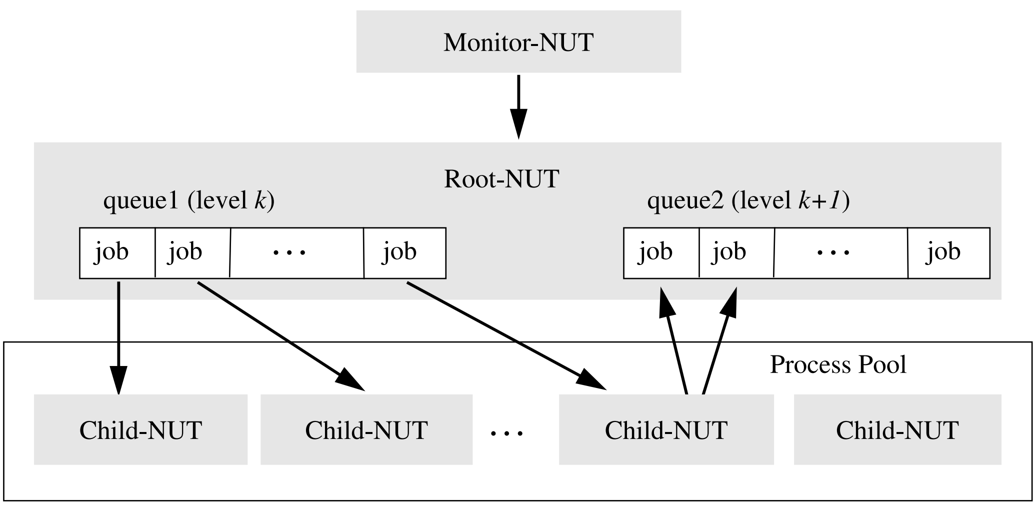


FIGURE 10. A Process Pool with Two Queues

Root takes the following actions:

1. making and administrating the process pool[1],
2. starting with root state, modifying root state and filling **queue1** by states of the first level,
3. sending jobs from queue1 to free processes, if there are no free processes, root waits until one process finishes its job,
4. when the **queue1** is empty and all the processes are free, fill queue1 by **queue2** and start searching in the next level (back to point 3),
5. If both queues are empty notify failure in search,
6. If answer is found, kill the process pool and notify the answer.

These were the main principles in our algorithm.

2.3.4 Design

Package design: The user interface is NUT window itself. By changing two classes (UsrParSearch and WorkSpaces) one can define her own BFS problem.

Message passing and administration of process pool are managed in other classes and the user don't need to care about them. But she can still change (through Scheme Editor) the configuration of PVM and she can also change the number of processes in the process pool.

```
%####################################################%
%       CONTROL STRUCTURES FOR PARALLEL COMPUTING
%             Demo package for PARSEARCH
%        Royal Institute of Technology, Stockholm
%#####################################################%
% Start PVM by selecting 'Compute ALL' under 'Scheme'
% after performing the following line:
      ClSchemeOpen('PVM');
% Create WorkSpaces for Root & Pool Processes(children)!
      ws := new WorkSpaces;
% Spawn Root:
      root := rnut_spawn(1,2,nil,'project/POOL.mem');
% Put & perform the WorkSpace for creating of the pool:
      rnut_pputwsr(root,1,ws.root_make_pool);
% Wait for the pool to be created and then start the search:
      rnut_pputwsr(root,1,ws.root_start_search);
% Wait until you get the object "answer":
      NutPrint(answer);
% Kill the root
      rnut_kill(root,1);
%Modify UsrParSearch &/or WorkSpaces,
%remake and Save the package.
%######################################################%
```

FIGURE 11. Text in the workspace of *monitor*

1. The ParSearch has been developed before the PAR_ENGINE, therefore the process pool is managed by class Pool.

The program starts in the main NUT process, the main NUT works as a **monitor** and spawns the search root (see Figure 13). This is a good choice because the user can still interact with the package during the root and process pool searching, furthermore if her search problem does not fit the BFS search, she can kill the root and modify her class without risking deadlocks or other crashing problem (see the workspace of **monitor NUT**, Figure 11).

Design of classes. In this part we present the important classes in the package and their main functionality:

- **UsrParSearch**:

After starting **monitor** NUT and loading the package, the search problem should be formulated in class UsrParSearch which contains following relations:

1. ROOTSTATE: gives the root state,
2. USRMODIFY: modifies the state of current nodes and produce children,
3. USRGOOD: becomes true if the state is the search target.

We believe that these relations are general enough to specify a large number of BFS problems.

- **ParSearch**:

This class has relations which call the UsrParSearch, it keeps information about path in the tree and a checklist to avoid starting duplicate nodes in the search tree. These relations are used:

1. MAKEROOT: gets all information about root,
2. APPENDPATH: completes path in the tree of the actual node,
3. ISGOOD: calls USRGOOD to check if the answer is found,
4. MODIFY: calls USRMODIFY to modify the state of the actual node and produces children, checks also duplicated child,
5. CHILDSINFO: makes the needed information to start children as a job.

- **Pool**:

This class is used by root process for making, administrating and terminating the process pool. It contains:

1. MAKEPOOL: creates the process pool and sends the scripts,
2. STARTJOB: sends a job to a process pool,
3. FREEPROCESS: note that a process is free,
4. EXIT: terminate the process pool.

- **NodeInfo**:

This class gives all the information a process needs to start proceeding a tree node.

- **List**:

This class manages updating of path and check lists, it contains:

1. CHECK: checks if a state is duplicate,
2. APPEND: extend the pathlist and checklist with the new state.

- **WorkSpaces**:

The root and pool processes have different scripts. This class contains texts which will replace the scripts of root & pool processes. User can change the number of pool processes by simply changing the scripts of the root. This class contains:

var
root_make_pool: text;
root_start_search: text;
childs_work_space: text;

- **PVM**

The configuration of hosts connected to PVM can be changed in this class, through modifying Scheme Editor.

This class has only one relation:

STARTPVM: adds the hosts which are presented in the Scheme Editor.

2.3.5 Node Communication

Communication between root and pool processes managed by combining non-blocking store and blocking fetch routines. Each node suspends, waiting to fetch an object from itself, the other process can now store the object in the suspended process and unblock it.

All the messages we discussed (no_more_child, answer, ...), has been implemented by using this method.

The only object which is being sent between root and **monitor** NUT, is the final search answer. Because the answer is unique, it is sent without blocking the **monitor** by storing in a non-blocking way (see Figure 12).

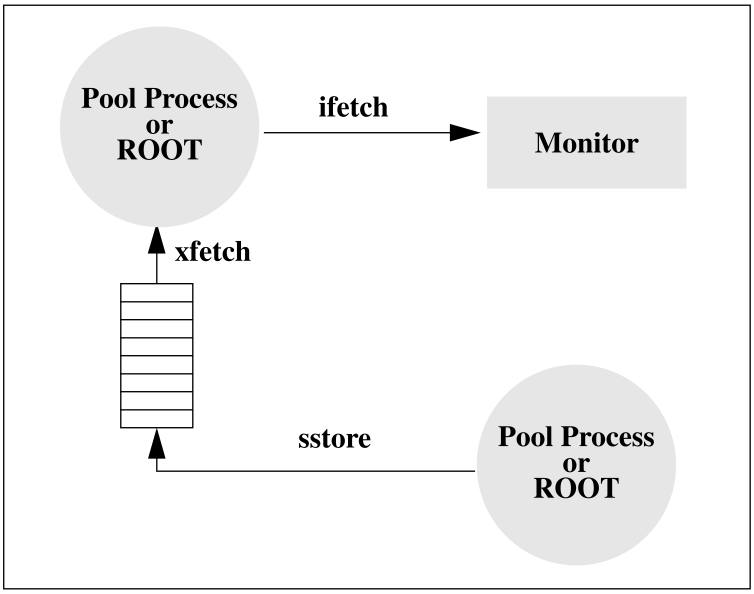


FIGURE 12. Node communication

2.3.6 Example

As a simple example we show a UsrParSearch which describes following problem:

- start with number 46 as the root state

- each node modifies by following three operations:

nystate = state * 2, nystate = state - 2, nystate = state + 2.

- search after state = 176

The UsrParSearch which describes this search problem can be formulated as follow:

```
super
        ParSearch;
vir
        outarr: array of any;
rel
        ROOT_STATE: -> out
        {
          out := 46;
        };
        USRGOOD: my_state -> out
        {
          if my_state == 176 -> out := true;
                  || true -> out := false;
          fi;
        };
        USRMODIFY: my_state -> outarr
        {
          outarr[1] := 2*my_state;
```

outarr[2] := my_state-2;
outarr[3] := my_state+2;
};

Figure 13 shows a typical situation on the screen while working with the package.

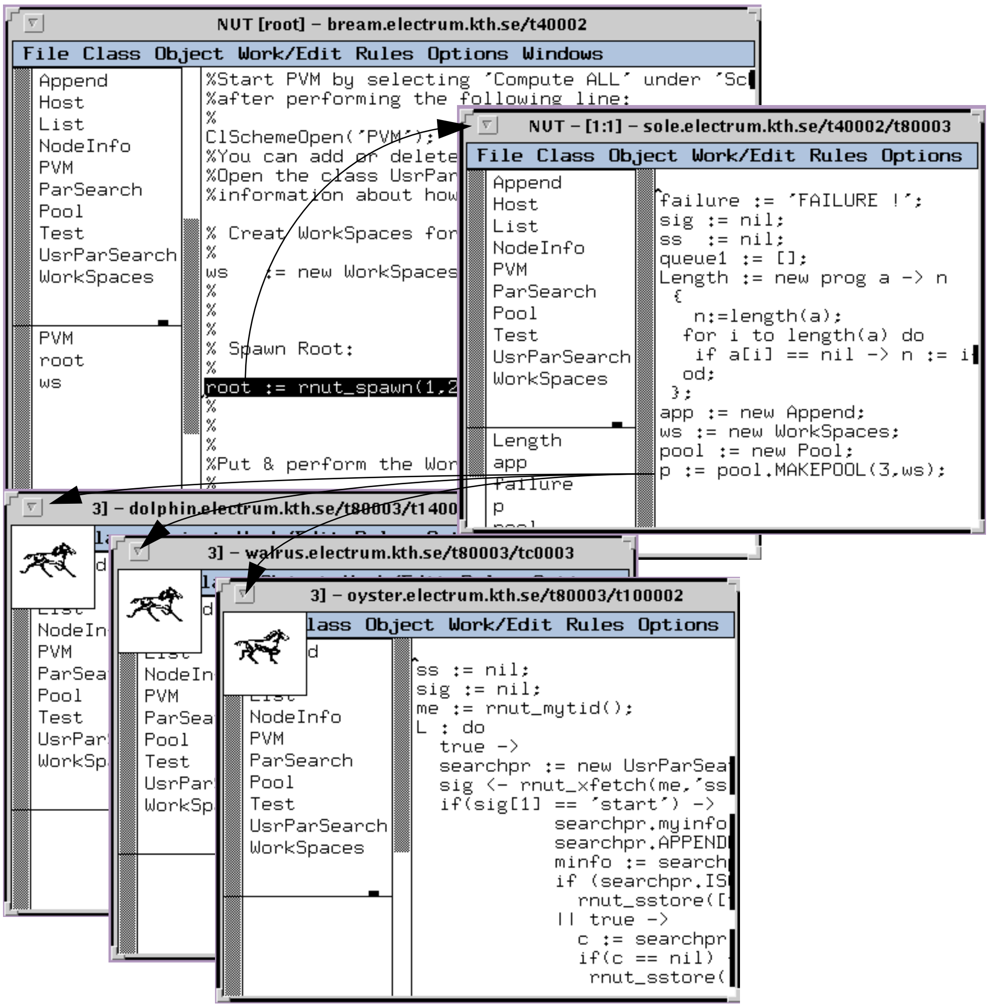


FIGURE 13. The PARSEARCH package and the process pool. Observe the *monitor* NUT on the top of the Figure, the root which has started in "sole" and, finally, see a process pool with three processes running in three different hosts.

2.4 PARARR

2.4.1 General Idea

In this part we discuss the control structure PARARR. Two more control structures have been developed during this phase:

- PARCLASS which can be used for parallel execution of independent tasks.
- PARPACK which is more suitable to be used in distributed application, using rNUT.

PARCLASS and PARPACK are discussed in chapter 2.5 and 2.6 respectively.

PARARR is implemented for parallel processing of elements in an array (element can be object of an arbitrary class). PARARR (created at the root process) sends each element of the array to a process (a worker) for proceeding, and collects the result from processes, see Figure 15.

2.4.2 Design

The PARARR is designed as a class in NUT.

The main relation in PARARR is:

RUN: X, C -> Y

- X is the input array, the array could contain **different instances of a class**, for example:

  X = [1,2,3,4], X = [[1,2],[3,4]] or X = ['parallel','computing'].
- C contains the name of the class which operates on an element, for example:

  C = 'increment', C = 'add' or C = 'reverse'.
- Y is the output array, using the values above PARARR gives:

  Y = [2,3,4,5], Y = [3,7] or Y = ['lellarap','gnitupmoc'].

The class C should declare an input and an output, as the interface to PARARR. This interface consists of two ***public objects***, **par_in** and **par_out** (Figure 15). This can be done by adding two relations in the class. See the following example:

```
class increment
    var
        a,b: num;
    rel
        a -> b { b := a+1;};
        par_in = a;   % PARARR passes the input to the class
        par_out = b; % PARARR gets the result from the class
```

The alternative is to declare C as a subtask (as in PARPACK), but we have realized that it would complicate the simple specification of C.

Now we can see how it works (Figure 14), PARARR creates an object of the class C (in a remote NUT process) and passes the input (X[i]) to **par_in** and asks the object to compute par_out. This is done automatically by extending the initial script which will be sent to the remote processes with the name of the class. The computed result collects finally in Y. The initial script, which will be modified before sending to the remote processes, is:

cc := new;   % The class name will be appended here, before sending the script.
cc.par_in := input;
cc.compute(par_out);
rnut_sstore([rnut_parent()], 1, ''ss'', [id, cc.par_out]);

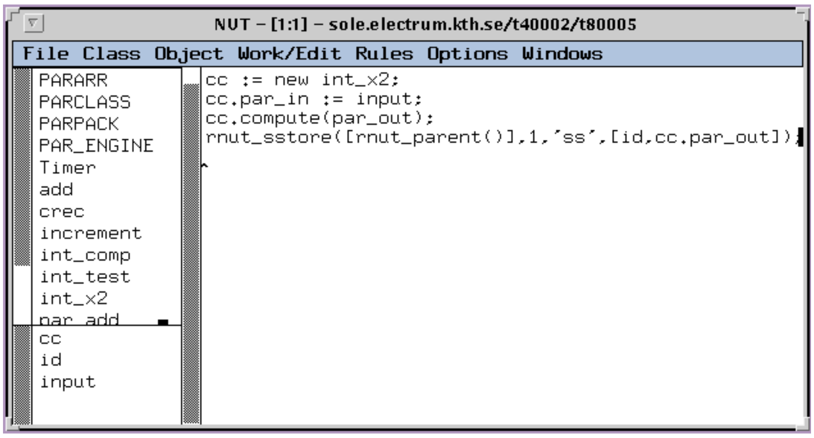


FIGURE 14. In order to compute a part of an integral, an object of class int_x2 is created (see the first line in the script), the input is passed to the object, the output is computed, and finally the result is sent back to the PARARR (at the root process).

Note that par_in could contain more than one input, but PARARR has just one input and one output, it's up to user to collect the inputs together in one element.

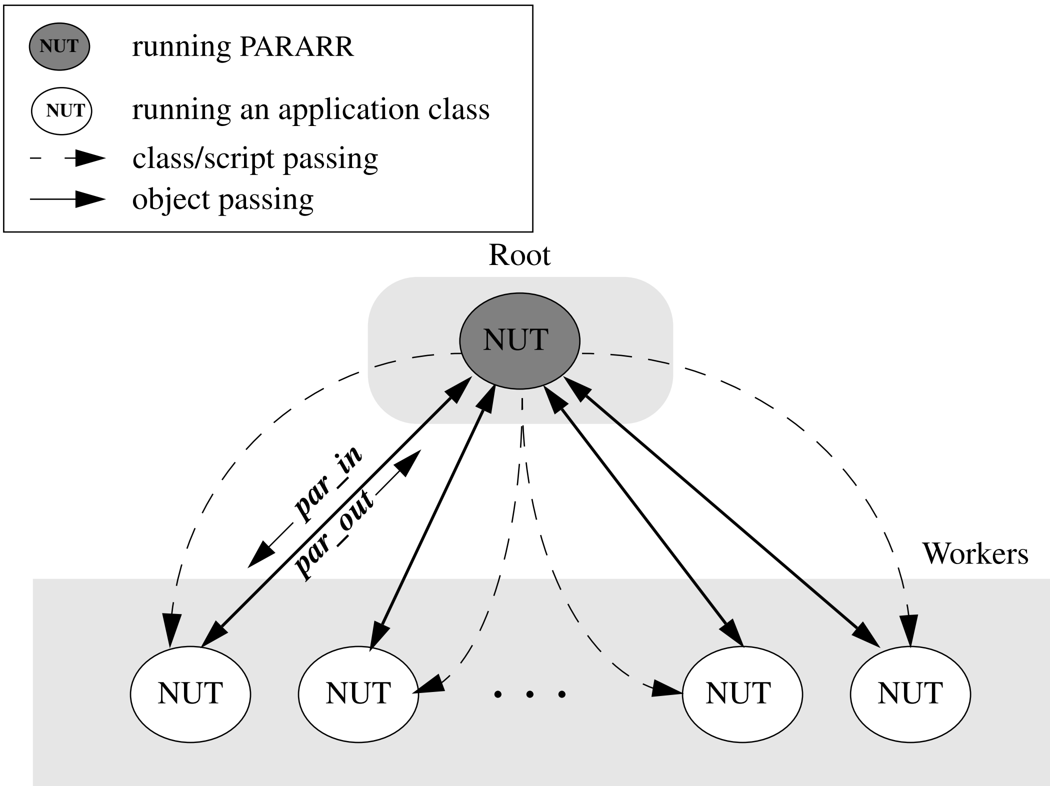


FIGURE 15. The communication pattern of PARARR

Instead of having two extra relations in the C and modifying the script, it is possible to modify the class text before sending it to the remote processes. But we couldn't use this approach because the text processing functions didn't work properly at the time of developing this class (see appendix 4.4.2).

PARARR can be simply used as a subtask (see the integral example), and therefore parallel problems can be formulated and computed in a nested way.

The administration of PVM and the processes are managed by using PAR_ENGINE.

Following methods are used in the PARARR:

- **RUN**:

  As we discussed before, this method takes the arrays of inputs and the name of the class which performs the computation on the inputs. It spawns n remote processes using **pool_manager**, and sends the inputs to processes and modifies script (by using the method MAKEWORKSPACE). Then it waits for the results, by using RECIEVE, and collects them into Y.

- **MAKEWORKSPACE**:

  Modifies the script to be sent to remote processes by appending the class name into the initial script.

- **RECIEVE**:

  Fetch the result of computation from a remote process.

2.4.3 Example

In the following example, we shall compute an integral by dividing it into p subintegrals, each partition will be sent to a process by PARARR, used as a subtask inside class **int_test**.

The script to compute the integral is:

```
P := new int_test x0 :=0, x1:=1, dx :=0.00001, p:=4;
P.compute(res); %compute 4 subintegrals in parallel.
NutPrint(P.res);
```

And class int_test has the following relation:

```
rel
    INT:[PARARR|-X,C->Y] x0, x1, dx, p -> res
      {
    A := new array of any;
    for i := 1 to p do
        %divide the interval into 4 parts.
          A[i] := [x0+(((x1-x0)*i-1)/p),x0+(((x1-x0)*i)/p),dx];
    od;
    subtask 1(A,'int_x2',B); %compute subintegrals in parallel using PARARR.
```

```
res := 0;
for i to length(B) do
        res := res + B[i];
od;
     };
```

Note that by using PARARR , we have hidden all aspect of process control and inter-process communication. This means that a set of problem, like the example above, can be implicitly parallelized.

2.5 PARCLASS

2.5.1 General Idea

The main idea here is to compute independent tasks in parallel. If the problem is decomposable into a set of independent subproblems, these subproblems can be solved in parallel. In PARARR we solve the same task with different inputs, but in PARCLASS we solve different tasks in parallel.

2.5.2 Design

PARCLASS is designed as a class in NUT.

The main relation here is the same as PARARR:

RUN: X, C -> Y

- X is the input structure, the structure could contain different instances of different classes, for example:

  X = [1,[1,2],'computing'].
- C contains names of the classes which operates on each element, for example:

  C = ['increment','add','reverse'].
- Y is the output structure, using the values above PARCLASS gives:

  Y = [2,3,'gnitupmoc'].

Here we have the same restriction for classes, they should contain two relations which declares the input and output of the class. In addition it should be a matching between the elements and the classes (between X and C).

PARCLASS can also be used as a subtask, each class can use a PAR-class as a subtask, this fact gives us possibility to define parallel problems in a nested way.

The only difference with PARARR is the RUN method which gets an array of class names. Objects of different classes will be sent to the remote processes.

2.6 PARPACK

2.6.1 General Idea

PARPACK is more proper for distributed application using rNUT. As we mentioned in environment notes a remote process can dynamically load a new package. One can specify her own problem using classes distributed in different packages instead of copying all the classes in her own package.

2.6.2 Design

Because one may use packages which are in a package library, we can't use the same approach as in PARARR and PARCLASS, by forcing user to modify her class. The class should be declared here as a subtask giving the name of input and output.

The main relation in PARPACK is:

RUN: X, C, P -> Y

- X is the input structure, the structure could contain different instances of different classes, for example:

  X = [1, 11].

- C contains names of the classes (specified as an independent subtask in NUT[1]) which operates on each element, for example:

  C = ['[circle|-S->r]', '[[fct|-n->f]'].

- P contains names of the packages which contain the classes, for example:

  P = ['/afs/it.kth.se/misc/packages/NUT/nutpack/tutorials/first.mem',
  '/afs/it.kth.se/misc/packages/NUT/nutpack/tutorials/secound.mem']

- Y is the output structure, using the values above PARPACK gives:

  Y = [0.564333, 3.99168e+07].

The input structure, the subtask definition and the names of packages should be carefully matched (X, C and P).

It is important to mention that the first goal in PARPACK is to use the classes in other packages without loading them into the current working package. In this context, it is not of primary concern to perform tasks in parallel, but to perform a task in a remote package.

The following two methods are different from those in PARARR:

1. The general form of a specification of an independent subtask is

[IC |- X -> Y]

where IC is the name of a defined class, and X and Y are lists of component names - the lists of the input and output parameters of the subtask.

- **RUN**:

  Same as PARARR, but from the subtask specification (in C) the name of the class as well as in- and output names are parsed. Then the initial script is modified by using the parsed names and finally, the package names (P) are used to load the desired package when spawning the processes.

- **PARSE**:

  Used for parsing the class name and the name of input and output from C.

2.6.3 Example

We take the same example described in 2.6.2. There are two classes, first class is located in the first.mem package and computes the radian of a circle giving the surface of it. The second class is fct located in secound.mem, this class computes the factorial of the input number. In Figure 16, see the root process and remote processes, each loaded by a remote package and proper inputs. The idea of making virtual packages in NUT can be seen in this Figure. Instead of keeping all the classes in the class library of the working package, one can dynamically use the classes distributed between different packages. In order to use this approach, one needs information about where these classes exist, their name and functionality. This is an important issue in OO, where code understanding is key to reuse of the existing software.

In appendix 4.1, we have proposed and implemented some experiments with OO software recovery in the NUT system. The proposed architecture, named SoftSpy, uses principles described here, combining with message passing by scripts in rNUT in order to retrieve information about a remote NUT package.

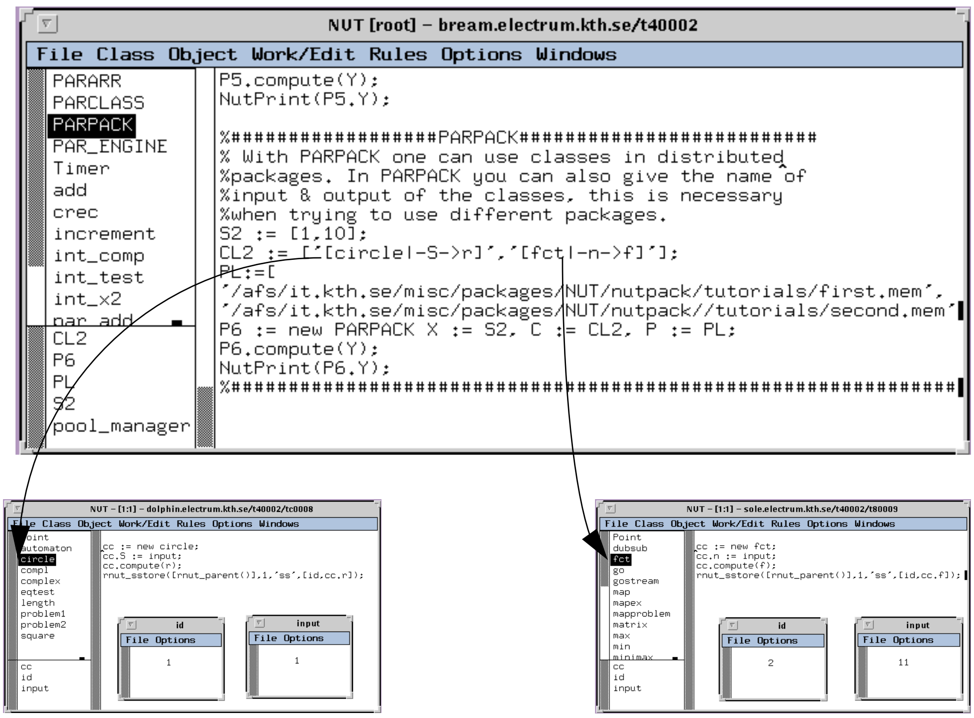


FIGURE 16. Using PARPACK for making a virtual package in NUT

2.7 PARDIF

2.7.1 General Idea

As discussed in the 2.4, the control structure PARARR provides communications only between the root and the workers. The more general control structure PARDIF creates a grid (one, two or three dimensional) of NUT processes with a communication pattern illustrated in Figure 17.

2.7.2 Design

Each worker from the grid can communicate with its neighbours. PARDIF can be used specially for problems which can be parallelized using data partitioning in one, two and three dimensions. PARDIF object in the root process distributes objects of an application class to workers as well as a script for communication control. Each worker can communicate with the root process using par_in and par_out components as in the PARARR communication pattern. To communicate with neighbours, a worker can use up to twelve communication components in an object of the application class:

set_up, get_up, set_down, get_down,

set_left, get_left, set_right, get_right,

set_back, get_back, set_forth and get_forth.

The first four components are used to communicate in one dimensional grid, the first eight components in two dimensional grid and all the components are needed in three dimensional grid of workers.

Each object of the application class which is sent to a worker, is initialized by PARDIF in the same way as in PARARR, by using **par_in**. Workers can perform computations iteratively. The number of iterations and the grid dimension are specified in the PARDIF and application objects during their initialization. During each iteration, a worker can communicate with its neighbours through the communication components mentioned above (**set_up**, **get_up**, etc.). After each iteration, all workers send the results to the root through the par_out component.

2.7.3 Example

To demonstrate a usage of the PARDIF structure, we have implemented a two-dimensional Laplace equation solver with Dirichlet boundary conditions, using the simultaneous displacement method of Jacobi [CT 92]. This problem called the Dirichlet problem is formulated as:

$$\nabla^2 F = \frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} = 0 \qquad \text{(EQ 1)}$$

To solve this equation, the surface F is partitioned into an n by n grid whose elements represent the initial state of F. The value of F at the surface boundaries is a known constant. In each iteration, a new value of F is calculated as:

$$F_{x,y} = (F_{x-1,y} + F_{x,y-1} + F_{x+1,y} + F_{x,y+1}) / 4 \qquad \text{(EQ 2)}$$

In the NUT application, the grid of the surface is represented by matrix which contains $n \times n$ elements. The initial matrix is divided into partitions, each of which is processed by a separate worker. On each iteration, the workers exchange their boundary rows and send results of the iteration to the root process to display. Figure 18 illustrates a scheme of main data communication between workers and the root process.

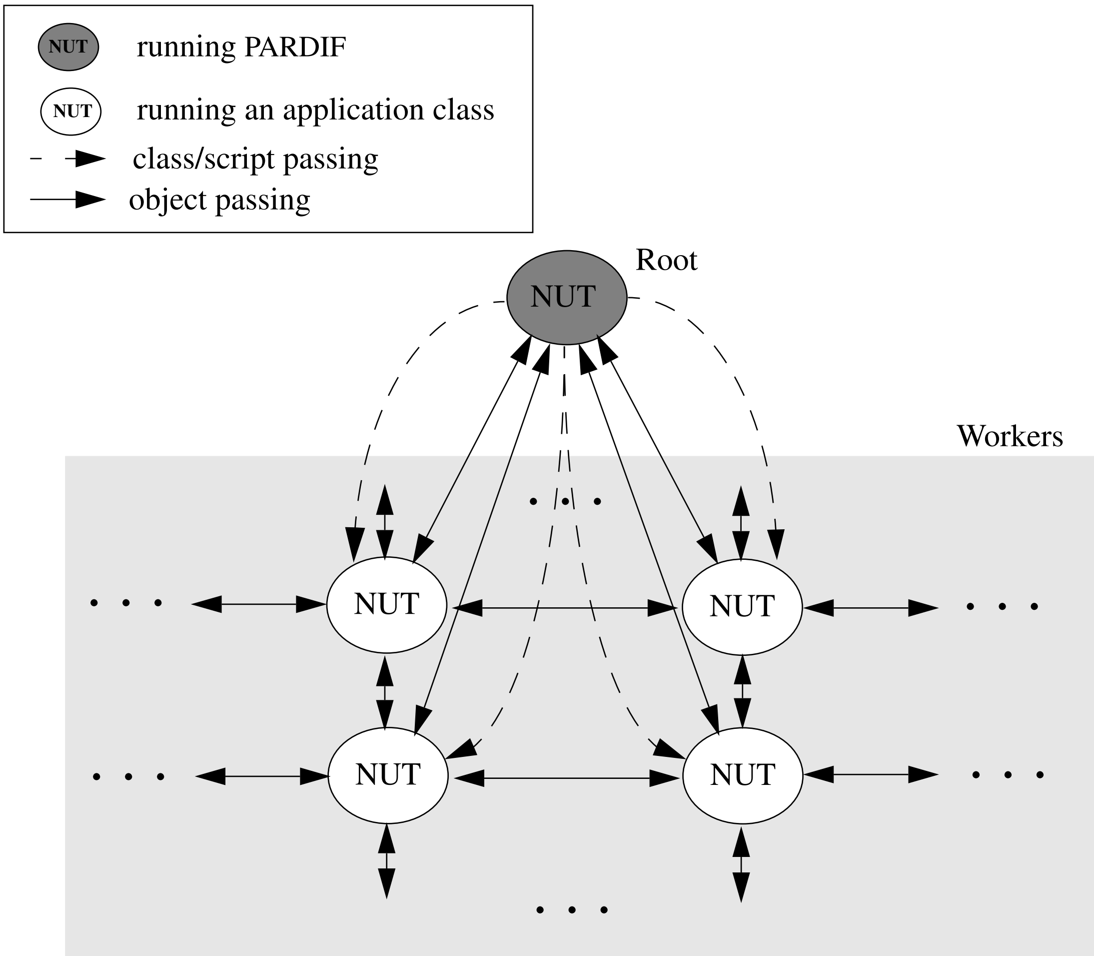


FIGURE 17 The communication pattern of PARDIF in two dimensions

The following NUT script is executed by the root process to calculate and display values of the matrix 100 by 100 during 1000 iterations, using four workers connected into one dimensional process grid:

```
% Creating objects:
pardif := new PARDIF;
M      := new DirMatrix;
matlab := new MATLAB;

% Initializations:
partitions := M.BuildPart(4);%partitioning the initial matrix.
max_iter   := 1000;
dim        := 1;

% Starting remote processes:
pardif.Exec(`Dirichlet', partitions, max_iter, dim);
```

```
% collecting results and sending to MATLAB for drawing
for i to max_iter do
m := pardif.result;
matlab.HotImage(M.BuildMatrix(m));
od;
```

The root process creates an object **pardif** of the PARDIF class, initializes the object **M** of the DirMatrix class and divides it into 4 partitions constituting an array **partitions**. These partitions are passed to the object **pardif** as the second parameter (in a method **Exec**). The root performs the method **Exec** in the object **pardif** in order to spawn four workers, create and initialize four objects from the application class Dirichlet (the first argument), and distribute them to workers for computing. The method **Exec** has two more arguments: the number of iterations and the dimension of partitioning. The result of each iteration is collected in the component **pardif.result** and passed to display using the object **matlab**. The interface between MATLAB and NUT is described in 4.2.

Each object of the application class Dirichlet has the following components for communication with its neighbours and the root:

par_in is used for fetching a partition of the initial matrix from the root,

par_out is used for passing the calculated value of the partition to the root after each iteration,

set_down, **set_up** (the output boundaries) are used for passing newly computed boundaries of the partition to the neighbour workers,

get_down, **get_up** (the input boundaries) are used for fetching boundaries of partitions from a neighbour workers.

The Dirichlet class is specified as follows:

```
class Dirichlet
  var
        my_part, par_in, par_out: DirMatrix;
        get_down, get_up,
        set_down, set_up: Array of num;
  rel
        par_out = my_part;

        Init: par_in -> my_part, set_down, set_up
        {
        my_part  := par_in;
        set_up   := my_part.GetRow(1);
        set_down := my_part.GetRow(my_part.m);
        };

        Iteration: my_part, get_down, get_up-> set_down, set_up
        {
        my_part.Iter(get_down, get_up);
        set_up   := my_part.GetRow(1);
        set_down := my_part.GetRow(my_part.m);
        };
```

The equation (EQ 2) is implemented as a method **Iter** in the class DirMatrix.

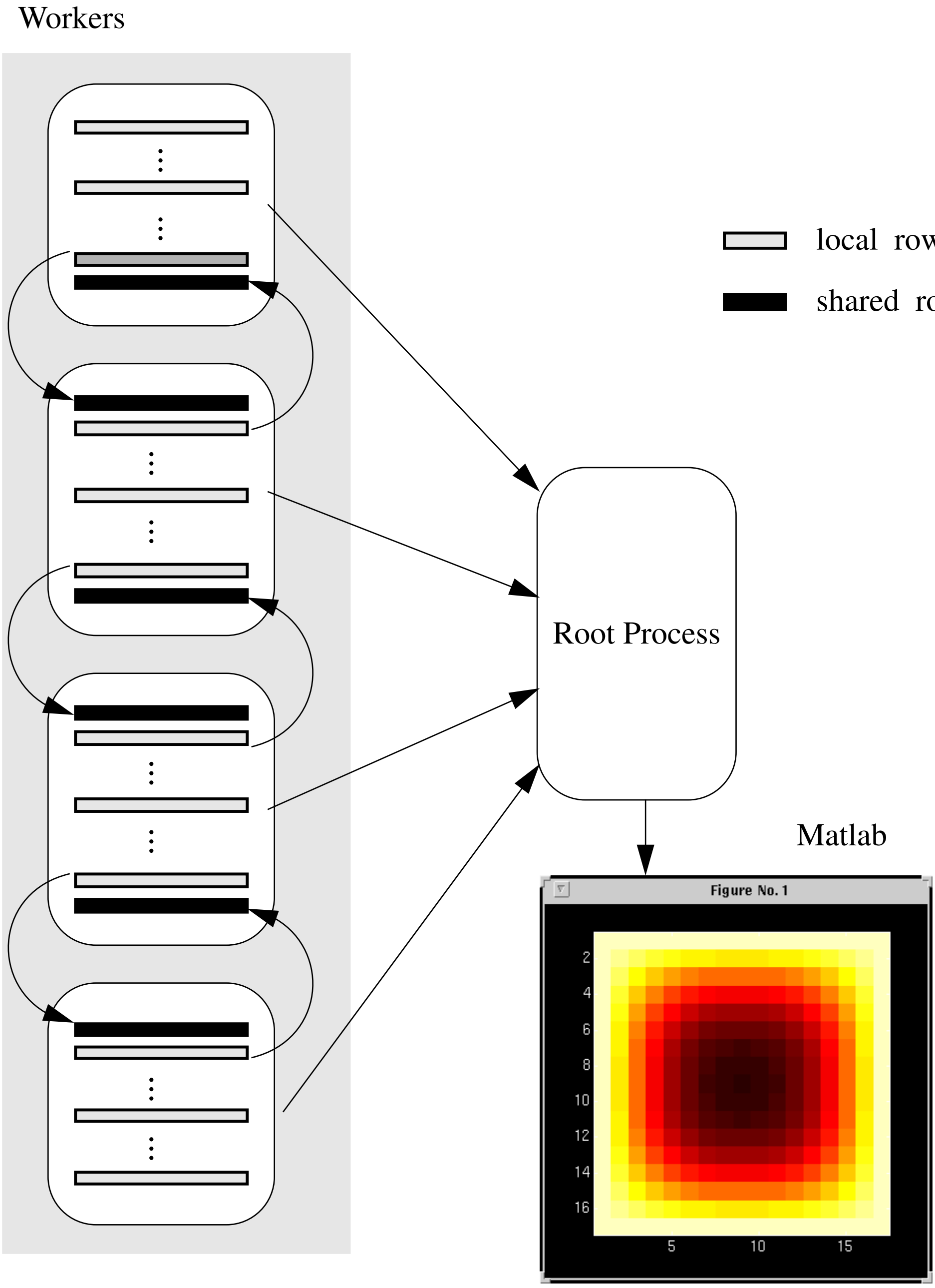


FIGURE 18. Solving the Dirichlet problem using PARDIF. In this example the matrix is divided into 4 submatrices. The boundary rows are duplicated by storing them in the neighbour process using remote s-store and local x-fetch.

2.8 Discussion & Proposal for Future Work

We have proposed and implemented an experimental platform for parallel programming in NUT, by using a set of control structures.

We have taken three types of control structures for experimenting.

PARSEARCH seems to be a good start for developing more structures of this kind, by extending functionality and implementing other search methods. The simplicity of PARSEARCH is a good example for the future extensions.The PARSEARCH hides all aspects of parallel search from the user in NUT. We believe that this class is general enough to specify a large number of BFS problems. More control structure as PARSEARCH can be developed for other kinds of search methods. More methods can be added to the current USRPARSEARCH for extending the functionality without losing the simplicity of expressing a search problem. For instance by applying an appropriate heuristic function to each node, for selecting the best promising ones, a best-first search can be managed. A good experimental area for future work can be combining the search methods using PARSEARCH and simulation.

PARARR is developed for parallel processing of elements in an array. PARCLASS is designed for performing tasks in parallel in those cases, where a problem can be decomposed into independent subproblems.

We have found following applications for PARPACK:

- Virtual package: Having a library of packages, one can combine a virtual package, by using the class libraries inside packages, using PARPACK. It increases the available Object Memory in NUT.
- Another application which has been inspired by PARPACK is SoftSpy. In the SoftSpy architecture, we spawn a remote process in order to analyse a package in NUT. The collected information can be then visualized in the remote process or can be sent back to the main process in the form of object messages. The communication is done by sending scripts and performing requests. This approach is discussed in more detail in the related appendix.

At last we took an example of solving Laplace equations using PARDIF.

Developing control structures needs good knowledge about the application domain. The idea is to develop control structures which are as general as possible. These control structures should also support a rich specification of a particular application without losing the simplicity of programming.

In the next chapter we examine how visualisation can reduce the effort of parallel programming in NUT.

3.0 Graphical Parallel Language in NUT

3.1 Introduction

Visual languages can be classified into three categories: (1) languages for processing visual information, (2) languages for supporting visual interaction, and (3) languages for *programming* with visual expressions, i.e., the visual programming languages [Bur+95].

The goal of visual programming languages are to use graphics and other visual techniques to allow the programmer to express the logic of a program, to see and trace the changes in program states and to give a greater overview about the program. What is important is that, in order to be considered a visual programming language, *the language itself* must employ some meaningful (i.e., not merely decorative) visual expressions as a means of programming [Shu88].

Visual object-oriented programming is an emerging area that combines the features of object-oriented technology with visual programming. The goal is to improve the quality and accessibility of information exchange between programmers and the computer, and to support developing programs that solve large and complex problems.

The NUT system supports visual object-oriented programming. Graphical representations of classes developed using the Scheme Editor of the NUT system constitute a graphical language. Program synthesis capabilities of the NUT system provide a possibility to implement rich semantics of graphical descriptions, including parallel computations.

The goal here is to develop a visual programming environment which reduces the effort of parallel programming in NUT by using the expressiveness of the graphical language.

3.2 General Idea

Visual and graphical features should be introduced to a parallel programming environment with respect to the existing tools and their functionality. In the NUT system, parallel programming can be supported by using distributed NUT processes, each remote process runs in a separate NUT environment consisting of a NUT package. The functionality of these remote processes and the communication structure is graphically represented using the Scheme Editor.

We have implemented some visual support for parallel programming in NUT.

In order to support graphical representation for processes and the communication structure of problem, we decompose the parallel programming process into two levels:

- **operational level**: At this level, a module to be executed in a remote process, is defined as a class (an object of this class will run on a remote process performing a set of methods and communicate with other processes).
- **structural level**: In order to realize the requested problem solution, the communication structure and the mapping of modules into the remote processes can be graphically described.

The operational level is programmed in the ordinary textual way. However there is some restriction on how a class should be written in order to satisfy the requirements in the *structural level*.

The graphical representation concerns the structural level, where the programmer can visually express the functionality of processes, and the interconnection between them which describes the flow of objects between processes. More visual information can be obtained, for instance the name of the related classes and the interface information about each process, i.e. the name of the ***public objects***, ***private objects*** (only those ***private objects*** which are used in the communication are visualized) and the *type* of communication operation *(x, s, i, u)*.

We have discussed before (in the environment notes) that a ***public object*** can be addressed by two components, the name of the object and the id of the rNUT process. The object names are given at the operational level. In order to access a ***public object***, we need to know the process id where the ***public object*** is placed. This information is coded at the structural level when a ***private object*** links to a ***public object*** of another process. Now the fetch/store operations can be graphically expressed as a link between one or more ***private object*** and a ***public object***. The port type determines the type of operation *(x, s, u, i)*.

By using these features, the distributed programming can be accomplished without concerning explicitly the fetch and store operations and process location, this information is derivable from the text of classes and the scheme representation of the program. By using this information, new classes are automatically generated. These new classes contain the *store* and *fetch* operations needed for communication between the remote processes.

3.3 Design

Here we discuss the operational level of programming. We describe *SOCKET*, which is the main component in our graphical model. Then we discuss in more detail the second level of programming, the structural level. Finally we mention the potential difficulties we met under implementation. More technical information can be found in appendix 4.3.

- *Operational level:*

A class in NUT contains variables, virtuals and methods. In order to make a class ready for the *structural level* (*SOCKETS*) following restrictions are enforced on a class:

1. A class must contain a RUN method. An object of this class will be sent to a remote process. The object will be activated by calling method RUN.
2. The variables which participate in the communication between processes should be declared as VIR (virtuals in NUT). These variables should be also a part of the external view (input and output parameters) of the RUN method.
3. The *type* of variables, described above, must be declared as a comment in NUT. The *type* will be parsed and used at the *structural level.*

Figure 19 shows two simple classes, a *Producer* and a *Consumer*, programmed for the *operational level*.

Producer

```
var
  t:     num;
vir
 send: num; %s
init
  t  :=  100;
rel
RUN: t -> send
{
      for i  to  t  do
          send := i;
      od;
      send := -1;
};
```

Consumer

```
vir
  rec: num; %x

rel
RUN: rec ->
{
    r := rec;
   do (r /= -1) ->
        NutPrint(r);
        r := rec;
   od;
};
```

FIGURE 19. Programming two classes at the operational level.

In the *Producer*, performing the RUN method in the ordinary NUT semantic, would finally assign -1 to the *send*. But in our programming model, it will produce an stream (note the type of *send*) of 101 values. These values will be passed to the *Consumer*, this is done by linking the *Producer* and the *Consumer* at the *structural level*.

The RUN method in the *Consumer* according to the ordinary NUT semantic would either print out nothing (rec = -1) or goes in an infinite loop (rec /= -1). In our model, it will fetch from *rec* into *r* as long as (rec /= -1) and print out 100 values passed by the producer.

At this level it is not clear yet whether the *store* and *fetch* operations are local or remote. This depends on the location of ***public object***. Both *rec* and *send* can be candidate as a ***public object***.

- *SOCKET*

Entities called *SOCKET* (see Figure 20) constitute the computational components of our programming model. The functionality of the *SOCKET* is defined by a class which is assigned to it. Each *SOCKET* represents an object of this class running either in a separate rNUT process or in the root process. The *SOCKET* is divided into an interface part which is accessible from outside (ports representing objects & a method "*RUN*") and an interior part which can be accessed only by *SOCKET* itself. The *SOCKET* interface consists of a set of input and output ports (10 ports are supported in the current implementation).

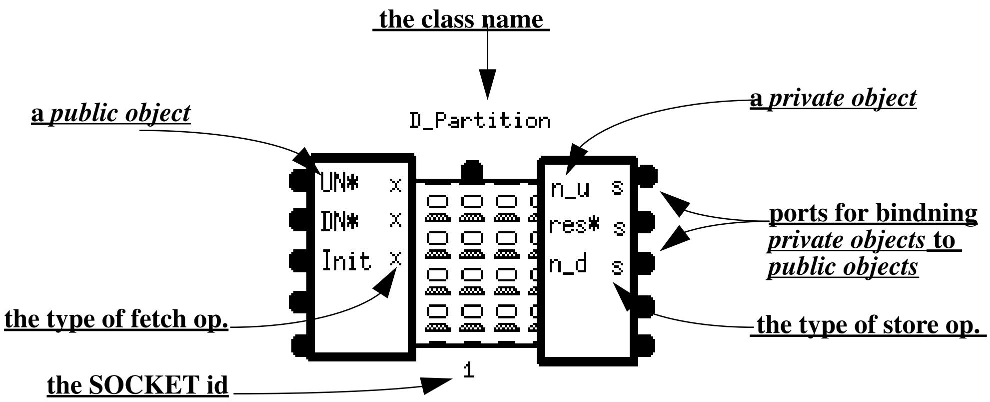


FIGURE 20. A SOCKET after programming, ready for binding with the other SOCKETS.

When a class is programmed, the class name is given to the *SOCKET.* After analysing the class, the names of the input and output objects and the *types* of *store/fetch* operation *(x, s, i or u)* are extracted from the class text and will be visualized. After programming at the *operational level,* i.e. writing the class to be assigned to the *SOCKET*, the *structural level* of programming can be done visually.

As we discussed in the environment notes, the ***public object*** can be considered as a shared object between two processes. The process of passing objects between *Producer* and a *Consumer* is showed in Figure 21, the *send* (a ***private object***) will be stored in the *rec* which is an ***public object*** in the consumer process (a ***public object*** is marked by a star), alternatively the *send* could be a ***public object*** and *rec* a ***private object*** in *Consumer.*

The *type* of an object is also visualised, a stream of *send* objects will be remotely stored and the *rec* is locally fetched by using *x type*.

The choice of the ***public object*** should be done with respect to the particular problem. Here it is more reasonable to have ***public object*** on the *Consumer* side, otherwise a remote *x-fetch* request will take two communication operations, one to send a *fetch* request and the other to receive a *reply.*

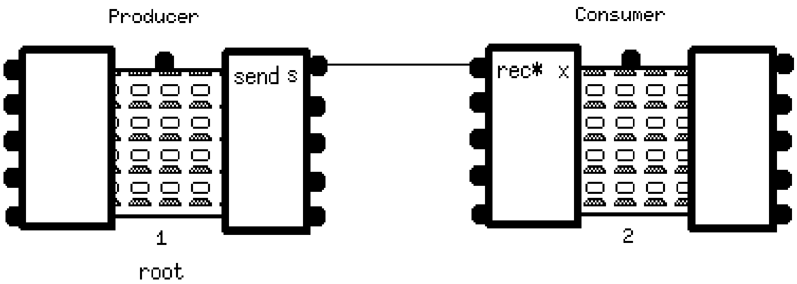


FIGURE 21. The *public object* is located in the consumer process.

As we mentioned, the names of ***public objects*** and ***private objects*** must be given at the *operational level*. When the programmer writes the classes, he should determine the names of these objects and the *type* of them. The difference between a ***private*** and ***public object*** can be made first at the *structural level.* The ***public object*** is marked by a star as it can be seen in Figure 21. The name of an ***public object*** given in the class text and the graphical notation gives all the information needed to locate a ***public object***.It is important to mention that these names are just for the sake of visualisation. One can use the same names for the objects in different processes. There is no risk for name conflict between different processes. For instance in the *Producer-Consumer* example one could use the same name (e.g. *buffer*) at both sides.

It is also possible to prescribe that a *SOCKET* should be executed in the root process. If more than one *SOCKET* are assigned to the root, they will be executed in the order of their *id* numbers.

This visual model is quite expressive to help the programmer to bind the processes and describes graphically the communication structure of the program.

- *Structural level*

The main activity at the *structural level* is describing the problem graphically using a set of *SOCKETS*. The programming environment is Scheme Editor and the message windows are used for collecting textual information. All the following steps can be done by choosing appropriate commands and answering in the message window.

When the problem is partitioned in a set of objects, an equal number of *SOCKETS* should be chosen. The next step is to assign a class to each *SOCKET.* When this step is done, the inputs and outputs of classes are automatically visualised. The location of ***public objects*** can now be determined by giving the name of the object. It is possible to determine if an object should be run in a separate process or in the root. Now the ports of the objects can be bound together. It is possible to bind more than two ports together, but there is always ***just one public object*** in a set of linked ports. ***This public object is the actual physical object which all the store and fetch operation are related to***.

After analysing the communication structure of the problem, the assigned classes will be automatically modified. Instances of these modified classes is sent to the spawned processes, a table over processes is broadcasted to all processes, and execution can begin (by broadcasting a message RUN to all objects).

The modified Producer-Consumer classes are showed in Figure 22, the class *COMMUNICATION* is added as a superclass. It contains methods for communication between processes (*STORE and FETCH)*.

Producer1

```
super
  COMMUNICATION;
var
  t:     num;
vir
  send: num; % s
init
  t  :=  100;
rel
RUN: t -> send
{
      for i  to  t  do
          STORE('rec',2,'s',i);
      od;
      STORE('rec',2,'s',-1);
};
```

Consumer2

```
super
  COMMUNICATION;


vir
  rec: num; % x

rel
RUN: rec ->
{
    r := FETCH('rec',2,'x');
   do (r /= -1) ->
        NutPrint(r);
        r := FETCH('rec',2,'x');
   od;
};
```

FIGURE 22. The modified classes after structural programming level, ready for mapping to remote processes.

As we can see in Figure 22, only ***public objects*** (*rec*) participate in the real communication operations. At the *structural level*, ***private objects*** (*send*) are not needed.

- *Potential difficulties*

The main difficulties we have met in the implementation of *GPLN* are related to functionality of the Scheme Editor and the text processing functions. We discuss these problems and possible solutions in details in the related appendixes (4.4.1, 4.4.2).

3.4 Discussion & Proposal for Future Work

Our approach breaks the distributed programming process into two level. If the programmer succeeds to break the problem into modules with clear communication structure, then our model offers a more clear view of how different parts of program are interacting, and a visual representation of the communication between remote processes.

When the SOCKETs use many ports and a problem is large, the benefit of having visual programming model is clear. We decided to visually program the Dirichlet problem discussed in 2.7. The experiment shows that changing and programming is much more convenient in visual case compared to textual. Also the code is smaller (see GPLN manual, 4.3).

Our visual distributed programming model is the first experiment in NUT. There are still many features to be added. We mention briefly some of them here:

- It is often claimed that one drawback of visual programming is the complexity of graphical representation, when problem becomes more complicated and the graphical components and connections become large and more sophisticated. One way to overcome this problem is breaking the program into modules with clear interface. Each module can be represented by a graphical block. By making block hierarchy, blocks and new components can be encapsulated in a new block. In general a visual language should support this feature. This is not automatically supported by Scheme Editor, but it can be done at the NUT level. We discussed this in 4.4.1.
- The restriction of having 10 ports for *SOCKET* should be reconsidered. One way is to introduce two new components, one for output-port and one for input-port extension.
- There is no graphical support for copying an object to a set of remote processes (broadcasting an object). Textually we can write:

  rnut_sstore(neighbours, length(neighbours), 'obj', new_obj);

  This means that new_obj will be stored in all the neighbours. This point can be also supported by using an extra component, *COPY.* In this case, the representation for the ports should be modified, in order to handle array of addresses.
- It should be possible to visually support a debugging mode, this is not difficult to add in our model. A separate process can be spawned for copying all values of ***public objects*** and for showing the debugging comments. This process can use graphical window for reflecting the state of remote processes. The implementation can be done easily because we modify the text of the classes, and we can easily add debug commands and operations inside the class texts.
- Implementing the control structures of the previous chapter by using *GPLN*. As an example, we have visually programmed the *Dirichlet problem* (see 4.3).

4.0 Appendixes

4.1 SoftSpy[1]

4.1.1 Description

We propose the ***SoftSpy*** architecture for performing experiments with object-oriented software recovery. When using the suggested architecture, different experiments can be arranged and various aspects of the code can be recovered.

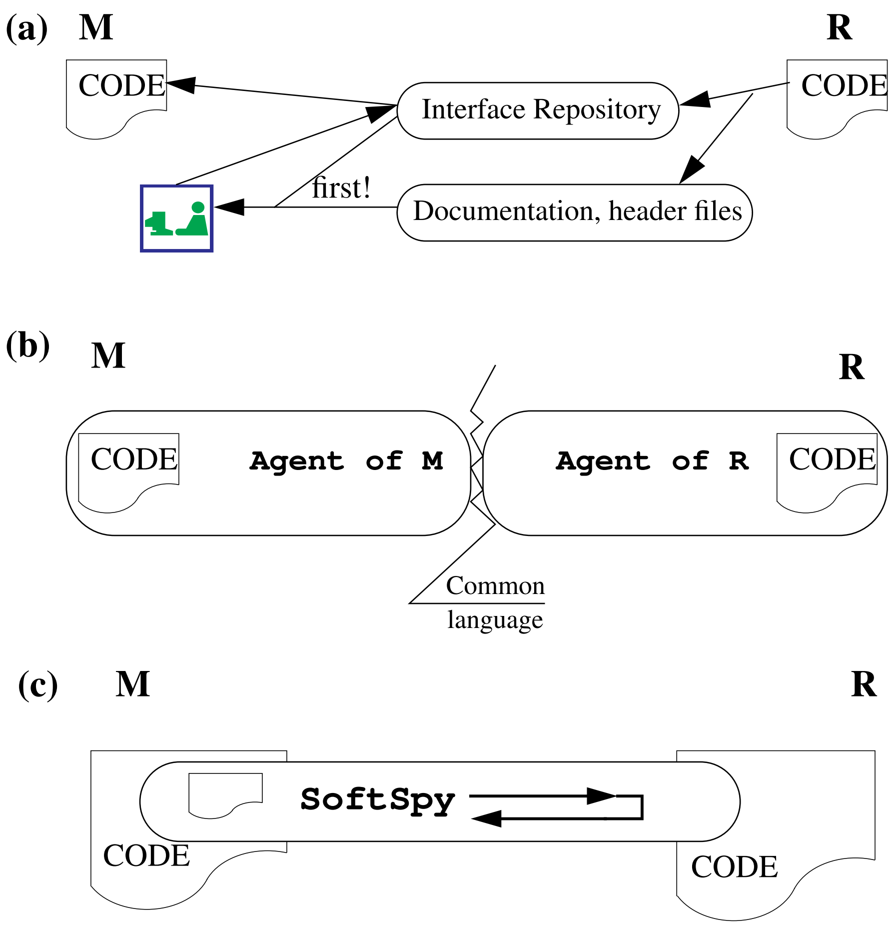


FIGURE 23. A comparison between some architectures which support the code understanding for reuse: (a) the most frequently used architecture for object-oriented distributed computing and object-oriented software development environments; (b) agent-based software architecture; (c) the *SoftSpy* architecture suggested by us. The letter R refers to the reusable system, or the system with reusable components. The letter M indicates the main system, or the system which is going to reuse the R system (or its components).

Figure 23 provides a comparison between our architecture and two other architectures.

1. The SoftSpy architecture have been proposed and developed in cooperation with Diana Sidarkeviciute and Enn Tyugu.

The letter R refers to the reusable system, or the system with reusable components. The letter **M** indicates the **main** system, or the system which is going to reuse the **R** system (or its components).

In **(a)** the general architecture of reuse employed in object-oriented distributed computing as well as in many object-oriented software development environments is shown. First of all, a programmer of the **M** system has to understand the functionality of potentially reusable components by looking at the interface repository as well as at documentation and header files. The interface repository contains the interfaces (attributes and methods) of classes available and also descriptions of data types. As we can see, understanding relies on the ability of the programmer to get the information quickly from the aforementioned sources. Some tools (for example, graphical representation of inheritance trees) are sometimes provided for the programmer, but they facilitate only the understanding of structural aspects of the **R** system code, and this is often not sufficient. Can the interface repository be extended? The results of the standardisation efforts concerning such kind of architecture (for example, CORBA suggested by the Object Management Group) shows that the language used for information repository must be very simple and describe only the structure of the reusable software. All behavioural aspects or any specific aspects (for example, evaluation criteria), must be understood from the documentation. In many cases, it may be impossible.

In **(b)** the general architecture of agent-based software [GK94] is shown. The research in agent-based software is emerging and promises a lot as it seeks to eliminate the participation of the programmer more and more. The software agents should be capable of communicating and exchanging any kind of information needed from the systems. Unfortunately, the problem of common language is very complicated and not solved yet. No understanding of the internal structure of agents, and no reuse of parts of agents are expected.

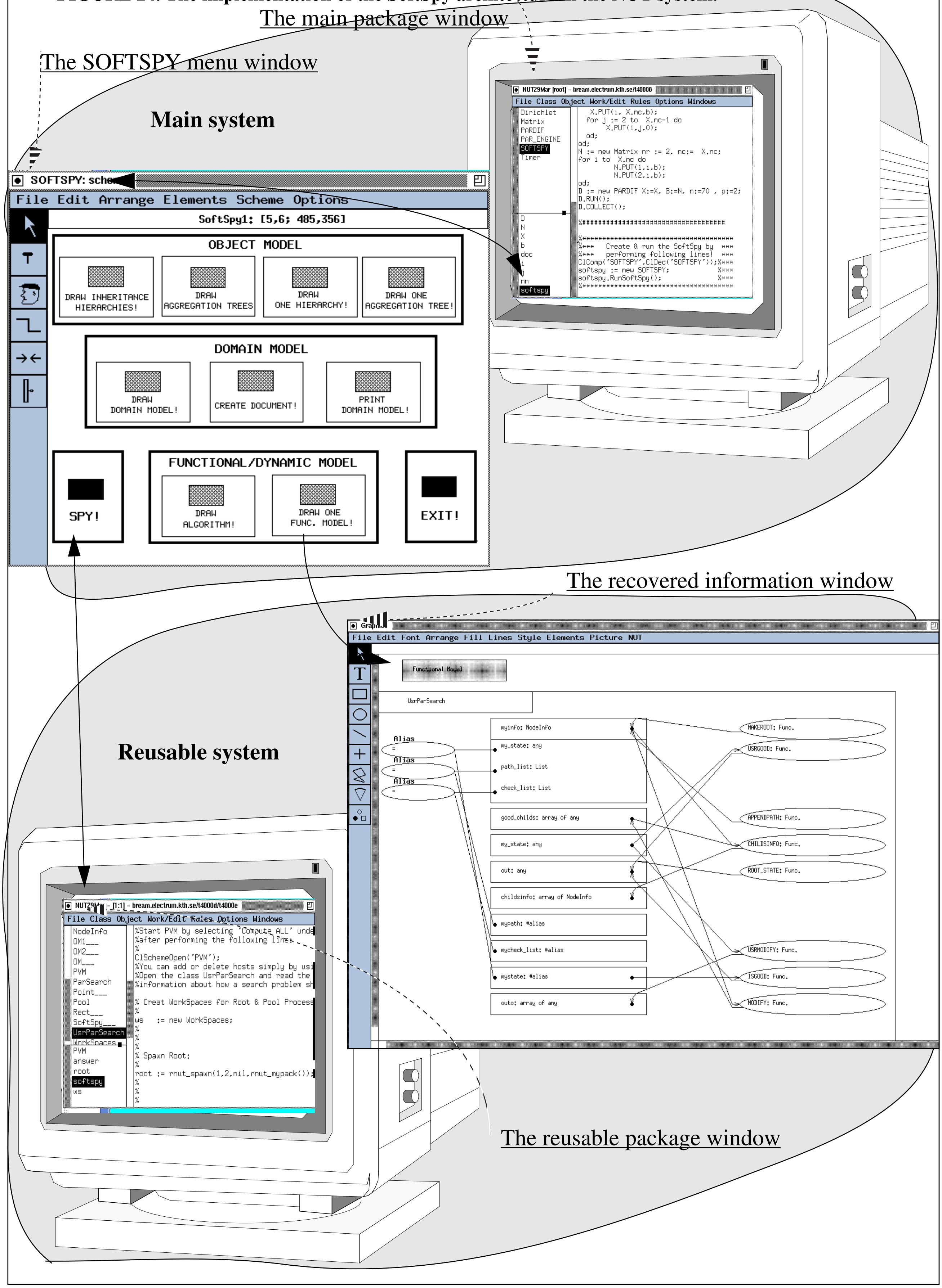
(c) presents our approach for supporting code understanding. We suggest a "golden mean". In other words, we support the programmer of the **M** system when he wants to get more information about the **R** system than is available in the interface repository. At the same time, we try to avoid introducing any common communication language. We assume that some program of the **M** system can execute some actions in the **R** system.Thus, we call this program ***SoftSpy*** and send it into the **R** system to execute certain actions and collect the information needed. Finally, we call it back.

We argue that the proposed architecture is general enough and flexible to be of help when arranging different kinds of experiments such as, for example, the evaluation of dynamical aspects of the components of a reusable system or recovering the dependencies between components.

4.1.2 Implementation

The implementation of the proposed *SoftSpy* architecture was done in the NUT system. The following components of the NUT system were employed in this implementation:
• A set of reflective functions which retrieve the information needed about the NUT package,
• A NUT graphical window which is used for the visualisation of the output,
• A NUT Scheme Editor as the SOFTSPY menu window,

FIGURE 24. The implementation of the SoftSpy architecture in the NUT system.



- A set of remote NUT functions for spawning and controlling processes as well as for the administration of object passing.

Figure 24 gives a snapshot of the work session in the NUT system when using the suggested architecture.

In the NUT system, a programmer works with a package which contains a set of classes, objects and a workspace for the execution of certain actions. A package window of the main package is shown in the upper right hand corner of Figure 24.

When the programmer wants to get some information about a potentially reusable package, he imports the class SOFTSPY into the main package. He creates an object of the class SOFTSPY and gets the SOFTSPY menu window (shown as the SOFTSPY menu window in Figure 24) displayed on the screen. Then, the programmer must specify the location and name of the reusable package and choose the action to be performed. When the reusable package has been correctly specified, a new session with the NUT system (as a separate process) is automatically started. The second NUT system has the reusable package (shown in another package window in Figure 24). Actually, the second process can be started with the closed window, but for demonstration purposes the reusable package window is displayed. Then the object of the SOFTSPY class is automatically sent to the reusable package, requested actions are executed and the results are shown (in the window of recovered information in Figure 24). The results can be also sent back to the main package. It is important that the actions with the reusable package are executed without interfering with the main package, e.g within a separate process.

We have implemented a general and flexible architecture. Future work will include the extension of functionality of SOFTSPY. Searching for keywords, and copying the classes into the main package, checking for name conflicts and version differences are some of the features which will be implemented in the near future.

4.2 MATLAB Interface

4.2.1 General Idea

For showing the results we have decided to use MATLAB as a graphical output. The idea of putting MATLAB with NUT has been always attractive. We have investigated two possibilities for connecting MATLAB to NUT.

In the first method, MATLAB and NUT are communicating via socket. MATLAB executes a MATLAB mex-program and waits for the NUT to call one of the socket ports. The NUT can now pipe the data to a particular socket address. MATLAB, receives the data from the given socket port and processes it, generally displaying graphics or processing data and sends back the results, by calling NUT.
The strength of this method is that MATLAB can run on different machines, and different NUT processes can use the same MATLAB in order to perform a common task.

The other possibility is piping MATLAB with NUT using the MATLAB Engine. By using the External Interface Library of MATLAB, it is possible to communicate with MATLAB through so called MATLAB Engine. Once MATLAB Engine is created it is possible to exchange data, and sending requests to MATLAB for evaluating script. One drawback of this method is that MATLAB should run in the same machine.

We used the second approach for connecting MATLAB to NUT.

4.2.2 Design

The routines described in this appendix support following operations:

- Starting and exiting remote MATLAB process from NUT (using MATLAB Engine),
- exchanging data between NUT and MATLAB, the following data formats are supported:

  Numbers (integer and double), array of numbers (integers and double), string and matrix (real and complex),
- performing request for evaluation of a MATLAB command (or a set of commands),
- echo mode for showing data transfer between NUT and MATLAB, as well as command execution in MATLAB.
- combination of MATLAB and NUT programs in a flexible way.

The translation of data between NUT and MATLAB, was the main design difficulty. In MATLAB, all data are represented as matrices (even single numbers and strings are represented as matrices). In Figure 25 data representation of matrices in NUT and MATLAB are shown. These representations can be translated by using the C representation of matrix in MATLAB. The representation of MATLAB's matrix in C is a struct, see Figure 26. In order to map the matrix into NUT a predefined class Matrix is implemented in NUT.

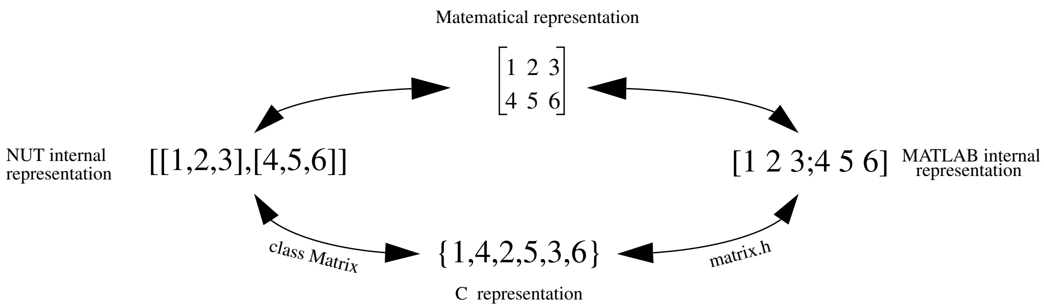


FIGURE 25. different representations of a matrix

This class has been proposed in order to meet the flexible requirement needed to map the matrix from MATLAB to NUT and vice versa.

- Class Matrix

  In order to map the matrices into NUT an object of this class must be passed as a parameter to the routines, The data will be collected in the proper components of class Matrix, depending on the type of matrix.

  All values in MATLAB are represented as matrices, and we use the same principle class Matrix to make this class representative for all data formats.

  The type component in the class shows the type of the returned value from MATLAB.

  Methods in the class Matrix can be implemented in different ways. But the mapping procedure of matrixes into array of numbers is done by using following relation:

  M[i,j] -> R(m*(j-1)+i) there **i** is the row number and **j** is the column number, and **m** is the number of rows.

  Both R and I are arrays representing real and imaginary part of matrix. Even strings are represented as a string in MATLAB, therefore we have a string variable in the class NUT (see Figure 26). In our model, it is allowed to implement different methods for processing the matrix in NUT. The restriction is that the class components represented in the Figure 26, must exist in the right order and the matrix should be represented as an array of columns (see the relation above). An implementation of class Matrix can be found in the class libraries.

C representation of Matrix in MATLAB

```
typedef struct matrix {
    char name[mxMAXNAM];  /* name */
    dummy;                /* type: 0 - matrix, 1 - string */
    int   m;              /* row dimension */
    int   n;              /* column dimension */
    Real  *pr;            /* pointer to real part */
    Real  *pi;            /* pointer to imag part */
    int   dummy6,dummy7,dummy8;
} Matrix;
```

class Matrix in NUT

```
class Matrix
var
  type:  text; %'r'->real, 'c'->complex, 's'->string
  m:     num;          % row dimension
  n:     num;          % column dimension
  R:     array of any; % real part
  I:     array of any; % imag part
  S:     text;         %string
```

FIGURE 26. Matrix data representation in MATLAB and NUT.

4.2.3 Discussion

The idea of connecting MATLAB to NUT is more than using MATLAB as a graphical output. NUT supports declarative programming, in particular, program synthesis and planning. A class in NUT can be used as specification for a computation model. The NUT synthesis a program using the information coded in the external view of methods and constraints. This is a powerful technique for automatic construction of new program out of the codes and specifications of existing programs. In this context, MATLAB's computational power can be used inside the methods wherever it is proper. In other words, the best scenario will be combining NUT's structural synthesis of programs with MATLAB's computational power. In this framework using NUT's Scheme Editor for high level graphical specification of programs and MATLAB for high level graphical output seems to be a promising platform for developing models for different engineering and scientific applications.

4.2.4 Future Work

Several possibilities exist to improve the usage of MATLAB together with NUT:

- Designing a set of classes in NUT, using the MATLAB commands for performing different tasks, which are so related that can be collected into a class. For instance, class Graphic and so forth.
- It is interesting to investigate the possibility of calling NUT from MATLAB. In this case MATLAB can call NUT for synthesizing and planning programs. It is possible to call NUT from C programs, and it should not be any difficulty to call NUT from MATLAB by using mex-files in MATLAB.
- MATLAB supports design of graphical user interfaces. By using NUT Scheme Editor it is possible to represent the different components in the graphical user interface (like buttons, menus, ...) as icons. By using these icons a graphical user interface can be graphically designed, and Scheme Editor can generate the M-file for the interface. It should be also possible to call NUT methods as call back functions, instead of MATLAB commands.

4.2.5 MATLAB Interface Routines

This chapter contains the routines used in MATLAB interface.

matlab_start

Start remote MATLAB process.

Synopsis

```
r := matlab_start();
```

Return values

r The routine returns 0, if starting fails;

1, if starting succeeds;

2, if MATLAB server is already opened.

Description

matlab_start starts a remote MATLAB process, by calling the MATLAB Engine library.

Example

```
%start the remote MATLAB.
r := matlab_start();
```

Errors

The routine ***matlab_start*** returns ***0*** in error situations, the routine displays a window to notify the error.

See Also

matlab_exit, matlab_put, matlab_perform

matlab_exit

Close down MATLAB server.

Synopsis

```
r := matlab_exit();
```

Return values

r The routine returns 0, if shutdown fails;

1, if shutdown succeeds;

2, if MATLAB is already closed.

Description

matlab_exit closes the remote MATLAB process, started by ***matlab_start***, ***matlab_perform*** or ***matlab_put***.

Example

```
%close the remote MATLAB.
r := matlab_exit();
```

Errors

The routine ***matlab_exit*** returns ***0*** in error situations, the routine displays a window to notify the error.

See Also

matlab_start, matlab_put, matlab_perform

matlab_put

Put a value in matlabs workspace.

Synopsis

```
r := matlab_put(M, name);
```

Return values

`r`	The routine returns 0, if routine fails; 1, if routine succeeds.

Arguments

`M`	Object to be put. One of the following object types can be used: **num, array of num, text, Matrix** (see the description of class Matrix).
`name`	Text object specifying the name of the variable in MATLABs workspace.

Description

matlab_put puts a value in MATLABs workspace. The routine converts the object type into matrix type in MATLABs representation. If MATLAB is not already opened, the routine starts the remote MATLAB process.

Example

```
matlab_put(4711, 'a');
matlab_put([1,2,3], 'Arr');
matlab_put('MATLAB interface.', 't');
M := new Matrix m := 1, n := 3; M.PUTROW(1,[1,2,3]);
matlab_put(M, 'm');
```

Errors

The routine ***matlab_put*** returns ***0*** in error situations, the routine displays a window to notify the error.

See Also

`matlab_start, matlab_get_mat`

matlab_get_mat

Get a matrix from matlabs workspace.

Synopsis

```
r := matlab_get_mat(M, name);
```

Return values

`r`	The routine returns 0, if routine fails; 1, if routine succeeds.

Arguments

`M`	Matrix to be get. M is an object of type Matrix which should be passed to the function. The Matrix will be returned in the parameter object (see the description of class Matrix).
`name`	Text object specifying the name of the variable in MATLABs workspace.

Description

matlab_get_mat gets a value from MATLABs workspace. The value can be matrix or string. The routine converts the MATLAB variable into an object of type Matrix in NUT. The type variable in matrix shows the type of matrix as follows:

'***s***' -> string , '***r***' -> real , '***c***' -> complex (see the examples).

Example

```
matlab_put('MATLAB interface.', 't');
M := new Matrix;
matlab_get_mat(M, 't');
NutPrint(M.S);
...
matlab_perform('Arr = [1+2i,2,3+0.5i]');
matlab_get_mat(M, 'Arr');
NutPrint(M.R);
if M.type == 'c' -> NutPrint(M.I); fi;
```

Errors

The routine ***matlab_get_mat*** returns ***0*** in error situations, the routine displays a window to notify the error.

See Also

`matlab_start,matlab_put,matlab_get_text,matlab_get_num, matlab_get_arr`

matlab_get_text

Get a string from matlabs workspace.

Synopsis

```
str := matlab_get_text(name);
```

Return values

`str`	The routine returns a text object.

Arguments

`name`	Text object specifying the name of the string variable in MATLABs workspace.

Description

matlab_get_text gets a string from MATLABs workspace. The routine convertes the MATLAB string into a text object in NUT.

Example

```
matlab_put('MATLAB interface.', 't');
tt := matlab_get_text('t');
NutPrint(tt);
```

Errors

The routine ***matlab_get_text*** returns ***nil*** in error situations, the routine displays a window to notify the error.

See Also

`matlab_start, matlab_get`

matlab_get_num

Get a number from matlabs workspace.

Synopsis

```
r := matlab_get_num(name);
```

Return values

r	The routine returns an integer or double numeric object.

Arguments

name	Text object specifying the name of the numeric variable in MATLABs workspace.

Description

matlab_get_num gets a number from MATLABs workspace. The routine converts the MATLAB number into a numeric object in NUT. The number should be real.

Example

```
matlab_put(4711, 'a');
aa := matlab_get_num('a');
```

Errors

The routine ***matlab_get_num*** returns ***nil*** in error situations, the routine displays a window to notify the error.

See Also

matlab_start,matlab_put,matlab_get_mat, matlab_get_text, matlab_get_arr

matlab_get_arr

Get an array of numbers from matlabs workspace.

Synopsis

```
arr := matlab_get_arr(name);
```

Return values

arr	The routine returns an array of integers or doubles numeric object.

Arguments

name	Text object specifying the name of the numeric variable in MATLABs workspace.

Description

matlab_get_arr gets an array of numbers from MATLABs workspace. The routine converts the MATLAB matrix into an array of numbers in NUT. A matrix with one column or one row can be converted to an array using this routine. The matrix should be real.

Example

```
matlab_put([1,2,3], 'arr');
arr := matlab_get_arr('arr');
```

Errors

The routine ***matlab_get_num*** returns ***nil*** in error situations, the routine displays a window to notify the error.

See Also

matlab_start,matlab_put,matlab_get_mat, matlab_get_text, matlab_get_num

matlab_perform

Execute MATLAB statement.

Synopsis

```
r := matlab_perform(command);
```

Return values

`r` The routine returns:

0, if the command isn't passed successfully to MATLAB.;

1, if the command is passed successfully to MATLAB (however the command may be failed in MATLAB.)

Arguments

`command` Text object specifying the string for MATLAB to execute.

Description

matlab_perform sends a string to MATLAB for performing. The string can consist of a set of MATLAB commands or name of M-files. Note that this routine does not give any information whether the command performs successfully or not. In order to check this, ***matlab_echo*** can be used for seeing the results of execution in MATLAB.

If MATLAB is not already opened, the routine starts the remote MATLAB process.

Example

```
matlab_put([1,2,3], 'm');
matlab_perform('m = m''*m');
M := new Matrix;
matlab_get_mat(M,'m');
```

Errors

The routine ***matlab_perform*** returns ***0*** in error situations, the routine displays a window to notify the error.

See Also

`matlab_start,matlab_echo`

matlab_echo

Select the echo mode for showing data transfer and command execution.

Synopsis

```
r := matlab_echo(mode);
```

Return values

`r` The routine returns 0, if selection fails;

1, if selection succeeds.

Arguments

`mode` Integer numeric object specifying the echo mode of MATLAB process. ***mode*** should have one of the values:

Mode value	MEANING
0	no echo
1	echo the MATLAB text output
2	echo the MATLAB text output and data transfer between NUT and MATLAB

Description

matlab_echo selects the mode fro showing MATLAB text output and the data transfer between NUT and MATLAB. This can be used for debugging the NUT programs which use MATLAB.

Example

```
matlab_echo(1); %open MATLAB text output.
```

Errors

The routine ***matlab_echo*** returns ***0*** in error situations, the routine displays a window to notify the error.

See Also

`matlab_perform, matlab_put, matlab_get_mat`

4.2.6 MAT-file Routines

In MATLAB, it is possible to save the matrices currently in memory to a binary file called a MAT-file. The load command performs the inverse operation, it reads the matrices from a MAT-file on disk back into the workspace of MATLAB.

It is also possible to read and write MAT-files using the set of routines in the External Interface Library of MATLAB. These routines offers manipulating MAT-files from NUT without invoking MATLAB process. We have implemented a set of routines which can be used in NUT for store, read and deleting the matrices (also String, number and arrays) produces by NUT or MATLAB.

The routines used in MAT-file manipulation are described in the following pages.

mat_fopen

Open a MAT-file.

Synopsis

```
r := mat_fopen(MAT-filename, mode);
```

Return values

`r` The routine returns:

0, if there is no MAT-file with name MAT-filename, it cannot be created/accessed properly, it is already open, or the value of mode is illegal;

1, if opening succeeds.

Arguments

`MAT-filename` Text object specifying the name of the MAT-file to be opened.

`mode` Text object specifying the mode of opening.

Description

mat_fopen opens the MAT-file named ***MAT-filename*** in a fashion depending on ***mode*** as follows:

`'r'`	open an existing MAT-file at the beginning, for reading;
`'w'`	create a new MAT-file, or open an existing one at the beginning, for writing;
`'a'`	create a new MAT-file, or open an existing one at the end, for writing.
`'r+'`	open an existing MAT-file at the beginning, for update (both reading and writing).
`'w+'`	create a new MAT-file, or open an existing one at the beginning, for update
`'a+'`	create a new MAT-file, or open an existing one at the end, for update.

Example

```
mat_fopen(`DifEq.mat','a+;); %open DifEq.matfor updating.
```

Errors

The routine ***MAT_ehco*** returns ***0*** in error situations.

See Also

`mat_fclose`

mat_fclose

Close a MAT-file.

Synopsis

```
r := mat_fclose(MAT-filename);
```

Return values

`r`	The routine returns:
	0, if there is no MAT-file with name MAT-filename or it is not open;
	1, if closing succeeds.

Arguments

`MAT-filename`	Text object specifying the name of the MAT-file to be opened.
`mode`	Text object specifying the mode of opening.

Description

mat_fclose closes the MAT-file named **`MAT-filename`** previously opened with **`mat_fopen`**.

Example

```
mat_fclose(`DifEq.mat'); %close DifEq.matfor.
```

Errors

The routine ***mat_fclose*** returns ***0*** in error situations.

See Also

`mat_fopen`

mat_fget

Get named Matrix from a MAT-file.

Synopsis

```
r := mat_fget(MAT-filename, M, matrix_name);
```

Return values

`r`	The routine returns:
	0, if routine fails;
	1, if routine succeeds.

Arguments

`MAT-filename`	Text object specifying the name of the MAT-file.
`M`	Object for storing the matrix (or string).
`matrix_name`	Text object specifying the matrix (string) to be get.

Description

mat_fget gets the named Matrix (***matrix_name***) from a MAT-file (MAT-filename), and store it in object M. M is an object of class Matrix and should be passed as a parameter to the routine. The returned matrix can be real, complex or string (see ***matlab_get_mat***).

Example

```
mat_fopen(`DifEq.mat','r');
%read m23 from MAT-file `DifEq.mat'and store it into M.
M := new Matrix;
mat_fget(`DifEq.mat',M, `m23');
```

Errors

The routine ***mat_fget*** returns ***0*** in error situations.

See Also

`mat_fget_next, mat_fput`

mat_fget_next

Get the next matrix from a MAT-file.

Synopsis

```
r := mat_fget_next(MAT-filename,M);
```

Return values

`r`	The routine returns:	
	0,	if EOF;
	1,	if routine succeeds.

Arguments

`MAT-filename`	Text object specifying the name of the MAT-file.
`M`	Object for storing the matrix or string.

Description

mat_fget_next gets the next Matrix from a MAT-file (***MAT-filename***), and store it in object M. M is an object of class Matrix and should be passed as a parameter to the routine. The returned matrix can be real, complex or string (see ***matlab_get_mat***).

Example

```
mat_fopen(`DifEq.mat','r');%open DifEq.matfor for reading
M := new Matrix;
mat_fget(`DifEq.mat', M, 'm23');
mat_fget_next(`DifEq.mat', M);
```

Errors

The routine ***mat_fget_next*** returns ***0*** in error situations.

See Also

`mat_fget_next, mat_fget_dir`

mat_fget_dir

Get a directory of matrices in MAT-file.

Synopsis

```
dir := mat_fget_dir(MAT-filename);
```

Return values

`dir` The routine returns an array of text objects, containing the name of matrices in a MAT-file. In error situation, it returns ***nil***.

Arguments

`MAT-filename` Text object specifying the name of the mat-file.

Description

mat_fget_dir gets the directory of matrices (including strings) in a MAT-file (***MAT-filename***).

Example

```
mat_fopen(`DifEq.mat','r');
dir := mat_fget_dir(`DifEq.mat');
```

Errors

The routine ***mat_fget_dir*** returns ***nil*** in error situations.

See Also

`mat_fget, mat_fget_next`

mat_fdel

Delete named Matrix from a MAT-file.

Synopsis

```
r := mat_fdel(MAT-filename,matrix_name);
```

Return values

`r` The routine returns:

0, if routine fails;

1, if routine succeeds.

Arguments

`MAT-filename` Text object specifying the name of the MAT-file to be opened.

`matrix_name` Text object specifying the matrix to delete.

Description

mat_fdel deletes the named Matrix or string (***matrix_name***) from a MAT-file (***MAT-filename***).

Example

```
mat_fopen(`DifEq.mat','r');
%delete m23 from MAT-file `DifEq.mat'.
mat_fdel(`DifEq.mat', `m23');
```

Errors

The routine ***mat_fdel*** returns ***0*** in error situations.

See Also

`mat_fget, mat_fput`

mat_fput

Put named Matrix in a MAT-file.

Synopsis

```
r := mat_fput(MAT-filename,M, matrix_name);
```

Return values

`r`	The routine returns:
	0, if routine fails;
	1, if routine succeeds.

Arguments

`MAT-filename`	Text object specifying the name of the MAT-file to be opened.
`M`	Object to put (number, array of number, string to matrix).
`matrix_name`	Text object specifying the matrix (string) to be get.

Description

mat_fput puts the object M in a MAT-file (MAT-filename). ***matrix_name*** specifies the name of matrix in the MAT-file. M can be an integer or double object, an array of integers or doubles, a text object or an object of class Matrix. (see ***matlab_put***).

Example

```
mat_fopen(‘DifEq.mat’,’w’);
mat_fput(‘DifEq.mat’,’Dirichlet’, ‘progname’);
mat_fput(‘DifEq.mat’,1000, ‘MATRICES’);
```

Errors

The routine ***mat_fput*** returns ***0*** in error situations.

See Also

`mat_fopen, mat_fget`

4.3 GPLN Manual[1]

Here we explain technical and practical details about *GPLN*. We choose the Dirichlet problem as an example.

To start writing program using *GPLN*, divide the problem into parallel parts. Then find out:

1. The interface of each module,
2. which objects are needed to participate in the communication,
3. how many processes and which classes are needed.

- **Operational level**

When it is clear how to divide the problem into different parallel modules, one can start with declaring the objects which participate in the communication and the RUN method. This part gives the needed information for Scheme Editor to visualise the external view of the module.

The important notes are:

- The objects participating in the communication must be declared as *VIR* (virtuals in NUT). Other ordinary variables should be declared as *VAR*.
- The types of objects are given as a comment in NUT.

  example:

  vir

          buffer: num; %s

  which means that buffer is a numerical object fetched or stored by using *s-type* communication operation (stream).
- The main method in the class should be RUN. The execution begins by calling this method.
- All the vir objects should be a part of the external view of the relation *RUN*.
- The vir objects should be used as follows:

  An input vir always returns a value, therefore a value should **not** be assigned to it.

  An output vir should always get a value from an assignment.

- ***Example***

In the Dirichlet problem, we divide the problem into following components, here we mention briefly the vir and the RUN method and some parts of code for showing the communication parts, the source code can be found in GPLN.mem.

- *D_Distributer*: divides the matrix into n partitions and initialises the dimension of matrix, number of iteration and initial value for the remote processes.

1. Read the chapter about GPLN (3.0) before starting this part, see also PARDIF (2.7).

```
vir
        o1: any; %s   Initialisation for the first partition
        o2: any; %s   Initialisation for the second partition
        o3: any; %s   Initialisation for the third partition
        o4: any; %s   Initialisation for the collector
rel
        RUN:  nn, p, t-> o1, o2, o3, o4
        {
        ...
        o1 := X;    % store the initial submatrix  in D_Partition
        ...
        };
```

Note that nn, p, t is ordinary NUT variables.

- *D_Partition*: There are n (3, in this example) such classes which run in separate remote processes, each process operates over a submatrix, and gets the upmost row and lower row from its neighbour.

```
vir
        Init: any;       %x     initialisation.
        UN:  Matrix; %x     get the boundary row from upmost neighbour
        DN:  Matrix; %x     get the boundary row from lower neighbour
        n_u:  Matrix; %s     store the upmost row in the neighbour
        res:   Matrix; %s     store the result of one iteration in  D_Collector
        n_d:  Matrix; %s     send the lower row in the  upmost neighbour
rel
        RUN: UN, DN, Init, X -> n_u, res, n_d
        {
         ...
        X := Init;   %Fetching Init from D_Distributer in to X
        ...
        n_d := X.GETROW(X.nr);%store the last row in the lower neighbour
        ...
        };
```

X is a ordinary NUT variable, other components make the interface of the SOCKET.

- *D_Collector*: collects the submatrices and put them together.

```
vir
          Init:     any;      %x        for initialization of class
          rec1:    Matrix;%x        get the submatrix from process nr 1
          rec2:    Matrix;%x        get the submatrix from process nr 2
          rec3:    Matrix;%x        get the submatrix from process nr 3
          M_out: any;     %s         send the final result to Matlab_Graph
rel
              RUN: Init, rec1, rec2, rec3, M -> M_out
                    {
        ...
        r2 := rec2;    % fetching a submatrix from the
                        % D_Partition into r2
        ...
                    };
```

- *Matlab_Graph:* calls the MATLAB for imaging the final matrix.

```
vir
         Matrix: any;%x              get the final result of one iteration.
rel
        RUN: Matrix -> res
        {...};
```

- **Structural level**

Programming of this part starts with creating a new class (*Dirichlet* in our example) which uses the class *PLANNER1* as a superclass. *PLANNER1* is the interactive part of the new class. Open the scheme and select the desired number of *SOCKETS*. Now open the object under Scheme (see Figure 27).

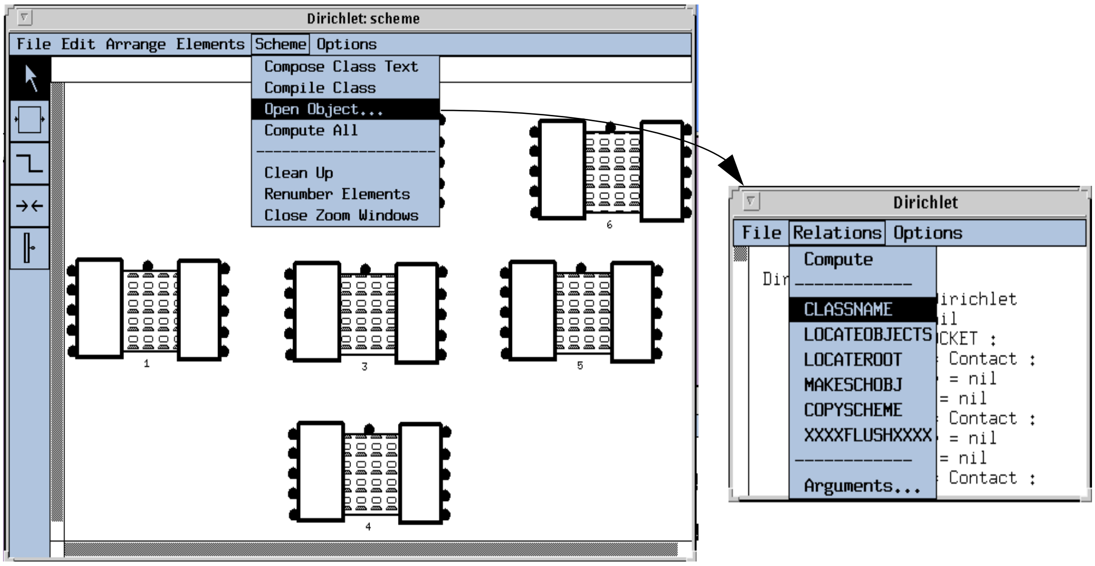


FIGURE 27. Select a number of SOCKETS and create the object, the programming can be followed by performing the methods in the object window of Dirichlet.

The relation menu of object can be followed by performing the methods after in the order. Following actions can be done:

- **CLASSNAME**: select this method to put the class name into the related *SOCKET*, a dialogue window will ask for names of classes. After performing this method, the names of classes, the interface objects and their *types* is automatically visualized (see Figure 28).

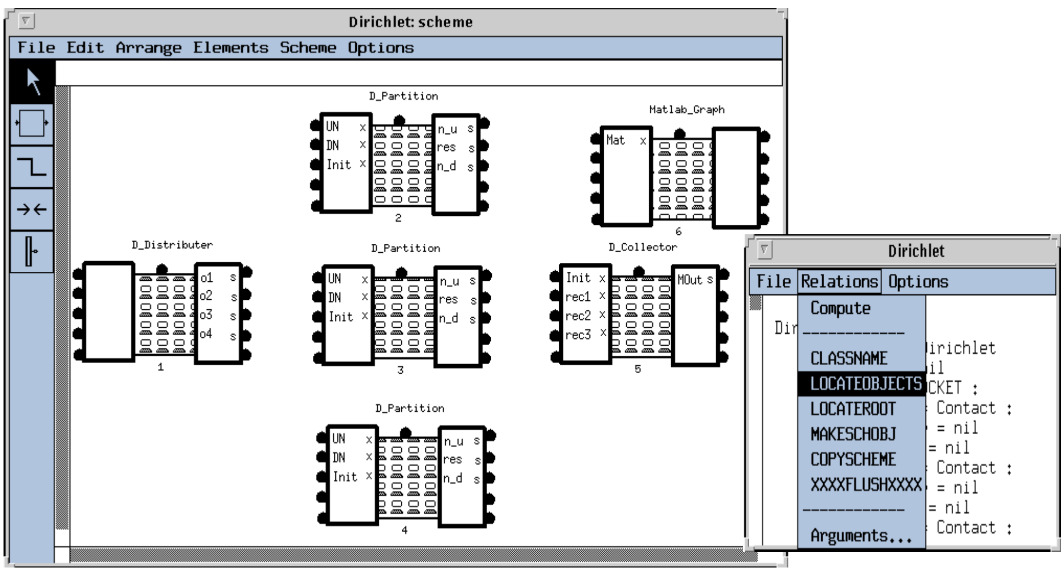


FIGURE 28. The name of the classes are given and the interface of SOCKETS are visualized.

- **LOCATEOBJECTS**: select this method to locate the ***public objects***. A dialogue window will ask for the name of ***public objects*** in each process (the name should be followed by '*' which indicates that the object is ***public****)*. The result is reflected in the scheme after performing.
- **LOCATEROOT**: By performing this method those *SOCKETS* which should be spawned outside the *root* can be selected (see Figure 29).
- **MAKEOBJECT**: All the collected information can now generate a new class with these data as initialisation. The reason of generating another class is discussed in the appendix about Scheme Editor (4.4.1). The method asks after the name of the new class (*Dirichlet_1* in our example).
- **COPYSCHEME**: This method copies the scheme of the first class, make sure that the scheme of first class (*Dirichlet*) is saved before performing *COPYSCHEME*.

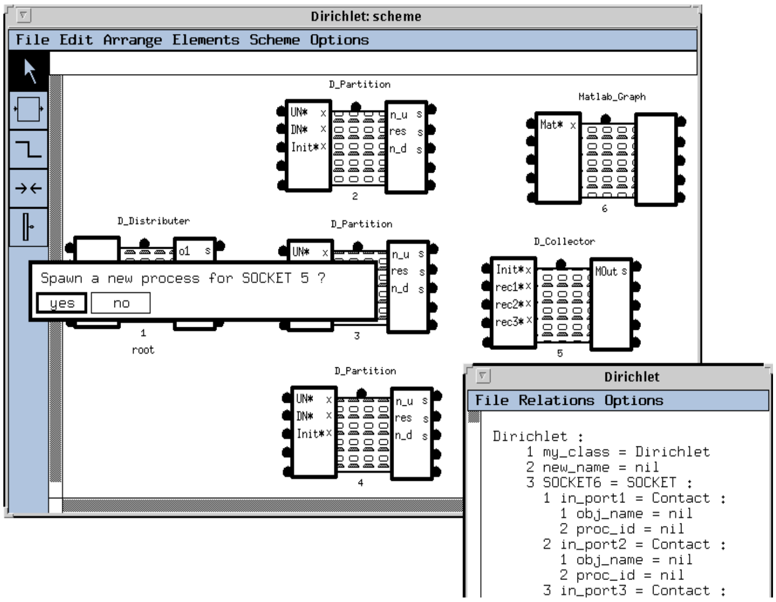


FIGURE 29. Locating the SOCKETS to be run in a remote process.

Now we can leave the *Dirichlet* class and the *Dirichlet_1* can be opened for drawing the communication structure.

After opening the scheme of *Dirichlet_1*, an object can be created. The superclass in *Dirirchlet_1* is *PLANNER2*. The following actions can be performed:

- **XXXXFLUSHXXXX**: start with performing this method which reflects the state of the object. In 4.4.1, we discussed why this method is needed.

  In the next step *SOCKET* ports can be connected. All the ports which represent an object must be connected. Each set of inter-connected ports must have only one ***public object*** connected. This means that each set of linked ports use only one shared object (***public object***).

- **PARSE**: This method generates new classes (by modifying the classes inside *SOCKET*). The class *COMMUNICATION* is added to all the classes as a superclass (see Figure 22). The new classes have the *SOCKET* id appended to their old names.
- **SPAWN**: spawns the remote processes and sends an object of each modified classes to a remote process. The remote NUT windows can be closed by using the **pool_manager** (see 2.2).
- **RUN**: this method activates the *RUN* method in all the remote processes (including the root. If there is more than one *SOCKET* assigned to the root, they will be performed in the order of their *SOCKET* id number).
- **EXIT**: kills the remote processes.

In the Figure 30 we see the final program and the produced image.

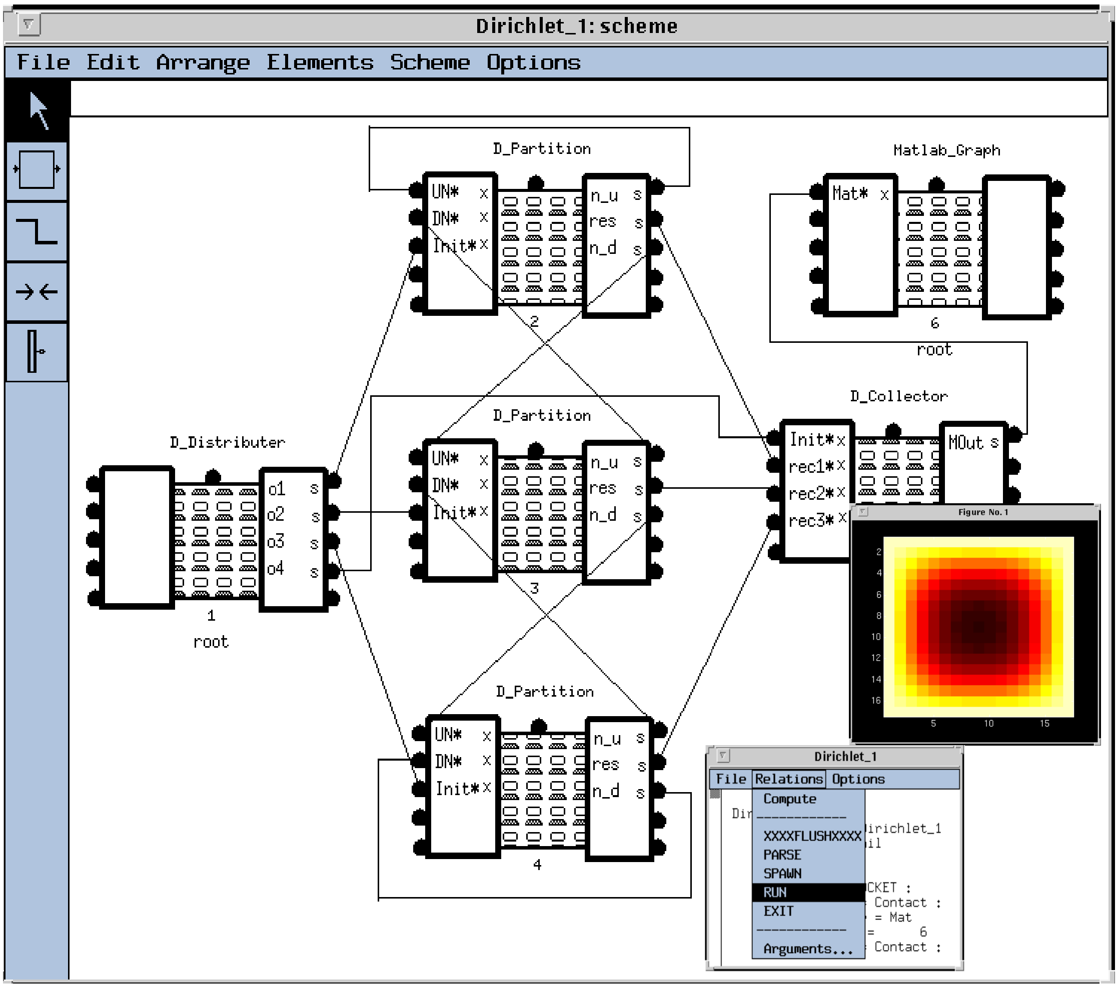


FIGURE 30. The final scheme, the object and the produced image by matlab.

4.4 Notes About the NUT System

4.4.1 Notes on Scheme Editor

The Scheme Editor of the NUT system is an extension of the Graphical Editor and is used as a tool for schematic (visual) definition of classes. In the current implementation, it is confusing to use Scheme Editor both for designing classes and using the same scheme for computations, based on objects created from class specification.

Some of the problems related to current implementation are mentioned below and the possible solution are implemented and discussed. However in the long run, these problems should be supported at a lower level.

1. *In the current approach, each scheme represents either a class or an object. It is not possible to have several schemes representing several instances of a the same class or having a scheme for a class and a separate one for representation of its object. It is desirable to have a class and its objects explicitly represented as separate schemes, and beside that* ***a safe way to save the schemes which represent the objects (in textual format, not in mem-format)****.*

P_C

```
super
  PLANNER;
% scheme begin
var
   SOCKET2: SOCKET;
   SOCKET1: SOCKET;
rel
SOCKET1.out_port1 = SOCKET2.in_port1;
% scheme end
```

P_C_OBJ1

```
super
  PLANNER;
% scheme begin
var
   SOCKET2: SOCKET;
   SOCKET1: SOCKET;
rel
SOCKET1.out_port1 = SOCKET2.in_port1;
% scheme end

init
   SOCKET1.id:=1;
   SOCKET1.root:=1;
   SOCKET1.cl_name:='Producer';
   SOCKET1.out1:='send';
   SOCKET2.id:=2;
   SOCKET2.root:=0;
   SOCKET2.cl_name:='Consumer';
   SOCKET2.in1:='rec*';
```

FIGURE 31. class P_C and a instance of it, constructed as a class (P_C_OBJ1, see the scheme in Figure 21).

This problem has been solved by using two methods (MAKESCHOBJ and COPYSCHEME) which constructs a new class scheme representing the current object scheme. **MAKESCHOBJ** modifies the class text and adds the initial values in the form of text, **COPYSCHEME** copies the scheme file for the new class. This approach has been used in implementation of *GPLN*, in which, an object has to be permanently represented in the package. In Figure 31, an instance of class *P_C (Producer-Consumer* in Figure 31), is represented as a new class *(P_C_OBJ1).* A better solution can be recording of initial values in a file (for instance at the end of the '*sch'* files), and possibly writing a class for managing this.

2. *Message Passing: Message passing is a fundamental concept in the OO. By knowing the name of an object, it should be possible to send a message to it. In the Scheme Editor we can express the relations by coupling the ports of objects together. But it is also desirable to have bindings for sending messages.*

   This is not supported in the current implementation, but there are different ways to support this, depending on the nature of a problem. The easiest way is connecting the communicative objects by using **self** (then each object have a reference to the other ones). This is not practical when the number of objects in the scheme is large.

   Generally, message passing between objects in the Scheme Editor can be supported by using a ***global object***. This ***global object*** can act as a message router (see Figure 32). It is easy to send messages to ***global objects***, a problem arises when objects at the same level (under ***top*** *level*) want to send a message to each other.

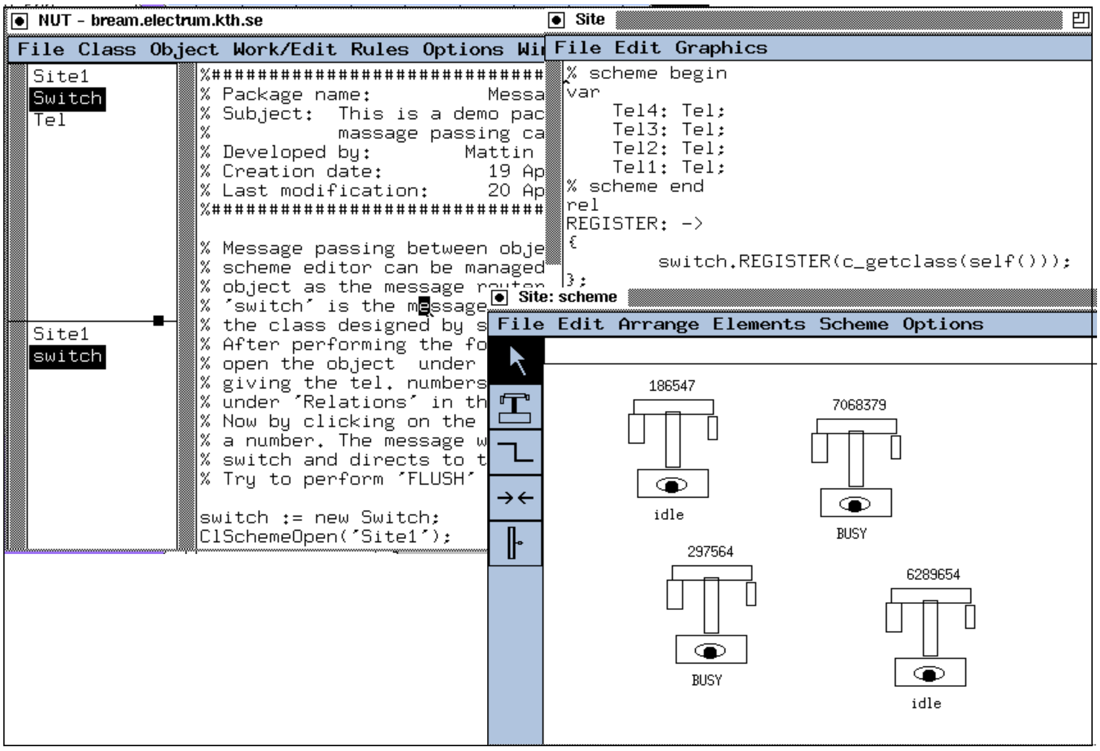


FIGURE 32. A demo for showing the message passing concept at the Scheme Editor level. The *switch* is a *global object* which accepts registration request from the other *global objects*. In this case the class *Site* designed by Scheme Editor contains a set of telephones. When clicking on a telephone we can call a number. The call request is passed to the *switch,* and directs to the related object.

In implementation of GPLN, we needed to locate the objects at the scheme level, and send different messages to them. We used the superclass PLANNER to manage this. This class locates other objects in the scheme class (at the same level) and by sending message, it can manage the tasks needed for our propose. But this approach is not general enough as we mentioned before.

3. *There is no graphical feedback of changes of the object state. For instance creating an object does not reflect the initialisation values in the graphic.*

   Considering this problem, it is enough to run a dummy method (with an empty body) from the object window. This will refresh the graphical screen to reflect the current state of the object. However, the problem of reflecting the state of the object at the run time still remains, this is especially desirable for simulation purposes.

4. *Scheme hierarchy: as we mentioned in 3.4, visual languages should in general support the graphical hierarchy, otherwise visual representation of large and complex problems becomes too sophisticated.*

   To support the graphical hierarchy in NUT, we find the following two approaches to be essential:

- **Single level grouping.** In this case grouping is done at one level at a time. In each scheme, groups from the same level can be represented. The interface of the groups are restricted (for instance two classes G*roupIn* and *GroupOut*). It is only possible to combine groups at the same hierarchy level. By choosing an image and icon for this scheme class, it can be used in a new scheme as a component, and group scheme hierarchies can be made by using this concept. In this case an scheme can be represented as a new icon without modifying the class text. Therefore making scheme hierarchies at single level grouping can be done completely without textual programming. However, the choice of *GroupIn* and *GroupOut* is very essential.

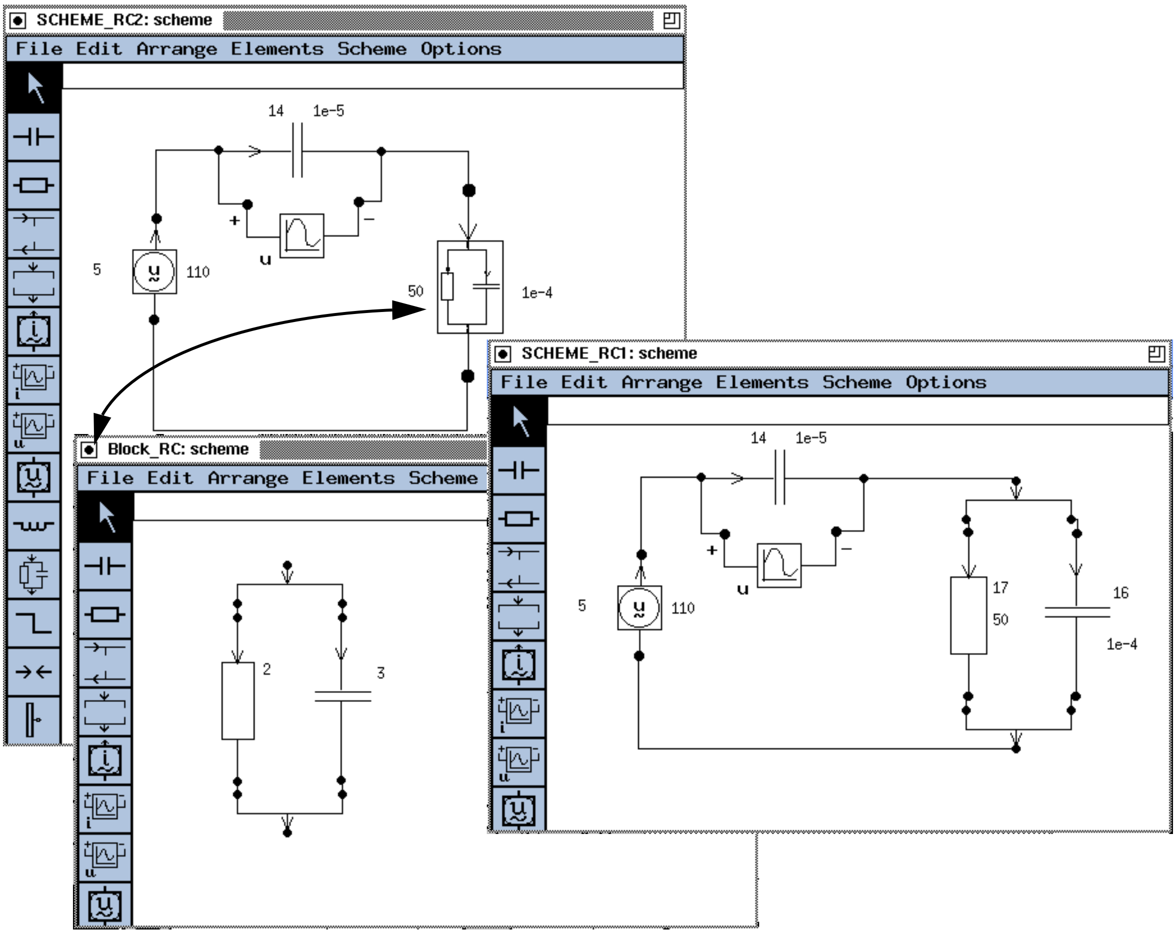


FIGURE 33. An example of group scheme hierarchy.

- **Multi level grouping.** In this case objects and groups of arbitrary graphical hierarchy level can be combined. It must be clear which ports can be combined and the text of the scheme classes must be modified.

  Here we show how the multi-level grouping can be done in NUT. In Figure 33 we see a simple case of grouping we have added to a student project. In the Figure you can see *SCHEME_RC1*. The RC is a good candidate to be encapsulate in a new class. The class can be designed by means of the Scheme Editor, and an image and icon defined. In the new representation of *SCHEME_RC1*, we can use the RC group instead of separate components. More complex circuits can be designed, using this method. This approach can be used in *GPLN* for reducing the complexity.

  By using group hierarchy one can use these groups which contains *SOCKETS* and eventually other groups and combine them in an arbitrary way (see Figure 34).

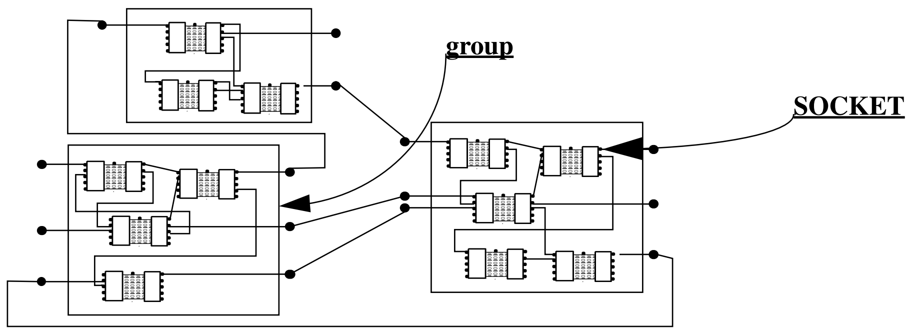


FIGURE 34. Making graphical group hierarchy in GPLN

In the current implementation of Scheme Editor, clicking on the group will give the class text. It is desirable to have class scheme at the first place if it exists.

4.4.2 Notes on Text Processing Functions

Classes serve two purpose in NUT. First, they act as classes in the sense of OO. Second they can be used as specification for a computation models or constraint networks. The ability of dynamically modifying classes becomes significant, when the model, described by the program, changes under the run time. A strength of the NUT system is in its reflective tools. By using a set of predefined and C-functions it is possible to get, analyse, modify texts of classes and finally recompile them. All these steps can be done in the run time[1].

As an example we mention experiments with intelligent network management [KT94], in which, changing network configuration model mirrored in the program by manipulating the texts of classes. We have also used this feature in most of PAR-classes, when the class text is extended with methods needed for communication, before sending to the remote processes.

Dynamic modification of classes can be done by applying the following steps:

- decompiling a class into its textual form,
- analysing the class using a set of reflecting functions, which finds its name, components, methods, etc.
- *modifying the text by using a conventional set of text processing functions,*
- recompiling the class using the modified text.

The text processing functions can manipulate a text string of at most 255 character, it means that the dynamical modification can be done only for small classes. This is not acceptable in *GPLN*, because the class is defined by user.

The following approach for solving this problem, has been used in *GPLN*.

The class text is written into a file (there exist *C*-functions for file manipulation), the text processing functions can be used now for processing a piece of text, read from the file. The modified text can be written back into the file. In the final stage, when the modified text is finally stored in the file, the whole text should be read back into a text object. In order to do that the routine described in 4.4.5.

Note that the **`c_fgets`** can not been used for our purpose, because it can read just one line, and it is not possible to put the lines together, again because of the limitation of text processing functions.

By using **`c_fgetext`** in combination with text processing functions, we can modify classes without limitation on their size.

1. This discussion is also valid for the “prog class” which represents the procedures in NUT.

4.4.3 Notes on Object Memory

Object Memory is used for storing and organizing memory in the NUT system. In NUT classes, methods, global objects and components of an object are all internal objects. The Object Memory is allocated statical[1] and therefore has a limited size. It has two limitations, the size of the allocated memory and the size of the object table. Each object has an entry in the table. The table has 16000 entries. These restrictions can cause two different problems. First, when the number of objects exceed the maximum number of entries in the object table, usually followed by the warning message 'Object table Exhausted'. The second problem arises when the allocated memory is completely used, although the number of objects have not exceeded the limitation of the object table. The latter case is not so crucial.

The main problem is the size limitation of the object table. This case happens mostly when too many object are created without any garbage collection. In the current implementation of most of the C_functions, each call to these functions produces one or more garbage and takes some entries in the object table. The garbage is produced when a temporary object is passed to the C_functions or is created inside the function. Although garbage collection can be done from the debug menu, this does not help in run time. These temporary objects must be collected at the run time to prevent garbage producing. In the case of rNUT the problem is more complicated and is discussed in 4.4.4.

It is also desirable to have unbounded Object Memory, because for larger packages and large applications, especially in the case of merging large packages, the size limitation is a great constraint.

1. The allocation is statical in order to make the Object Memory faster.

4.4.4 Notes on rNUT

The current implementation of object passing in rNUT should be modified. In the following we discuss the current implementation and the drawbacks of this approach. Then we propose a solution to the problems and discuss some new functions needed for more process control support in rNUT. Finally we describe a reflective function implemented using the rNUT library.

The object passing in rNUT is managed as follows (here we discuss remote store and local fetch, but this discussion is general for other kinds of object passing):

- The object to be sent is packed into the PVM send buffer.
- The destination process receives the store request and unpacks the object. Then, the object is stored in the Object Memory, and the object index in the object table is stored in a FIFO queue.
- The fetch operation will read the next object from the memory (by taking an object index from the FIFO queue).

In other words, the Object Memory of NUT is used as a buffer for storing objects, which are waiting in the queue to be fetched. This approach has many drawbacks:

- A process which gets many requests for store or fetch, can not perform anything else than unpacking the objects and storing them in the Object Memory, until the Object Memory is full.
- Because of the limited size of the object table, using it for storing temporary data is not a proper choice. The object table can easily exhausted.
- The objects are unnecessarily buffered at two levels. First in PVM and then in the NUT Object Memory.
- The buffered objects in the Object Memory, become garbage after fetching. This garbage can not be removed at the run time and it is the main reason that the object table, in the current implementation, becomes exhausted.

It was expected that rNUT would unbounded the Object Memory of NUT. Because a set of n collaborating NUT processes can use n times greater Object Memory than a single one. But because of the current implementation this feature can not be properly exploited.

We propose that in the future development of rNUT these concepts should be used:

- Objects should not be buffered in the NUT Object Memory, rather they should be kept in the PVM buffer.
- If the store or fetch request requires the object to be stored in a queue, the name of the object and the buffer id should be copied from the PVM buffer and should be queued. In this case the object is not unpacked. Instead of using Object Memory for buffering, PVM should be used for this purpose.
- When an object is needed to be fetched from the queue, the buffer id shows the location of the object in the PVM. After getting and unpacking the object from the buffer, it can be written directly over the previous one.

In this approach we have eliminated the need of storing the *public objects* in an intermediate object in the Object Memory of NUT. The *public object* exists in the PVM buffer. It will not be unpacked until it is needed to be fetched into a private object. By using this approach we increase the performance (because no additional time is wasted for duplicating the buffering in the NUT Object Memory), and beside that, no temporary objects and therefore no garbage are produced.

These following functions are also proposed to be implemented in the future development of rNUT, since these functions can't be supported at NUT level and should be implemented as a rNUT routine.

- **rnut_spawnh(nnut, mode, where, package)**

  As we mentioned in PAR_ENGINE, it is desirable to spawn n processes in m hosts. Although the philosophy of PVM is to select the most appropriate hosts, it's sometimes needed to prevent a host (like root host in our case) to participate in making a grid. This can not be supported by **rnut_spawn**.

  rnut_spawnh makes a grid of **nnut** processes in **where** which is a list of hosts.

- **rnut_grid(n), rnut_children()**

  A child process in rNUT can get tids of all the processes in its grid. But a process can't get the tids of its children. The motivation is that a process can spawn different grids and the problem is which tids should be returned? Therefore tids should be saved at the NUT level, because if we somehow lost the tids, the damage can't be recovered. **rnut_grid(n)** returns tids of nth grid spawned by a process.

  rnut_children returns all tids of children, although they may spawned in different grids (but all spawned by the same process).

  rnut_grid(n) returns tids of nth grid spawned by a process.

It is possible to get a class text, by using a reflective function. This routine can be used as a reflective tool for modifying the class. It is also desirable to get the script of the NUT package. The routine **get_ws** has been implemented for serving this purpose and is described in 4.4.5. This routine should be also classified as a reflective function.

4.4.5 Miscellaneous Routines

c_fgetext

Read the whole file into a text object.

Synopsis

`c_fgetext(filename) : text` or **`num`**
`filename : text`

Inputs

`filename`	the name of the file from where to read

Action

Reads the file named **`filename`** previously opened with **`c_fopen`** for reading until the end of file.

Returns

the read string, if reading is successful;
0, if there is no file named **`filename`**, or it is not open for reading

See Also

`c_fopen, c_fgets, c_fclose`

get_ws

Get a copy of the text of the Workspace from NUT.

Synopsis

`get_ws( tid )`

Inputs

`tid`	Integer numeric object specifying the tid of the NUT.

Action

Routine ***get_ws*** is used to get a copy of the text of the Workspace from the NUT specified by ***tid***.

tid is the tid of the NUT process (see the example).

Returns

the read script, if reading is successful;
nil, if reading is failed.

Example

nut_script := get_ws(rnut_mytid());

NutPrint(nut_script);

Errors

Routines return ***nil*** in error situations. In the open user interaction mode, on failure, routines display a window to notify the user about the error. In the closed user interaction mode routines print an error message to ***stderr***.

5.0 References

[AV94] B. Andersson, B. Volozh, "User interface of NUT". Teleinformatics/KTH, 1994 (Internal report).

[B+94] A. B. Beguelin, et al, "HeNCE: A Heterogeneous Network Computing Environment". Scientific Programming, Vol. 3, No. 1, Spring 1994.

[Bur+95] M.M.Burnett, et al, "Visual Object-Oriented programming, concepts and environments". Manning Publication Co, 1995.

[CH94] V. Cyras, M Haveraaen, "Programming with Data Dependencies: a Comparison of two Approaches". Proceedings of the 5th Nordic Workshop on program correctness, 1994.

[CT92] K. M. Chandy, S Taylor, "An introduction to parallel programming". Jones and Bartlett Publishers, 1992.

[Cyr88] V. Cyras, "Loop Program Synthesis Using Array Traversing Modules". Forschungsberichte Künstliche Intelligenz, Report FKI-93-88, Technische Universität München, Institut für Informatik, August 1988.

[G+94] G.A. Geist, et al, "PVM3 User's Guide and Reference Manual". ORNL/TM-12187, Oak Ridge National Lab, May 1994.

[G+93] U. Geuder, et al, "GRIDS-A Parallel Programming System for Grid-Based Algorithms". The Computer Journal, Vol. 36, No. 8, 1993.

[GK94] M. R. Genesereth , S. P. G.A. Ketchpel, "Software Agents". Communications of the ACM, 37(7), pp. 48-53.

[Int94] "The NUT libraries". Teleinformatics/KTH, 1994 (Internal report).

[Kop94a] U. Kopra. "Parallel Implementation of Synthesized Programs in NUT". Teleinformatics/KTH, 1994 (Internal report).

[Kop94b] M. Köpp. "Interoperability of NUT with C and Unix". Teleinformatics/KTH 1994 (Internal report).

[KT94] V. Koktas, E. Tyugu. "Knowledge-Based Techniques for Network Management". TRITA-IT-94;ISSN1103-534X;ISRN KTH/IT/R--94/09-SE.KTH, March 1994.

[Qui94] M. J. Quinn, "Parallel Computing". McGraw-Hill, 1994.

[Ric83] E. Rich, "Artificial Intelligence". McGraw-Hill, 1983.

[SGD+94] V.S. Sunderam, G.A. Geist, J. Dongarra , et al, "The PVM Concurrent Computing System: Evaluation, Experiences, and trends". Parallel Computing, Vol. 20, No. 4, April 1994, pp. 531-546.

[Shu88] Nan C. Shu,"Visual Programming". Van Nostrand Reinhold Co, 1988.

[Tho94] L-E. Thorelli, "The EDA Multiprocessing Model". Technical Report TRITA-IT-R 94:28, CSLab, Teleinformatics/KTH, 1994.

[TL95] L-E. Thorelli, D Liib, "A Language for Composing Program Modules". TRITA-IT-R 95:14, Teleinformatics/KTH, 1995.

[TU94] E. Tyugu, T. Uustalu, "Higher-order functional constraint networks". Constraint programming. NATO ASI Series F: Computer and System Sciences, Vol. 131. Springer Verlag, 1994, 116 - 139.

[Tyu92] E. Tyugu, "Algorithms of Artificial Intelligennce". KTH, 1992.

[Tyu94a] E. Tyugu. "The NUT system".Teleinformatics/KTH, 1994 (Internal report).

[Tyu94b] E. Tyugu, "Using classes as specifications for automatic construction of programs in the NUT system". Journal of Automated Software Engineering, v. 1, 1994, pp. 315 - 334.

[UKK+94] T. Uustalu, U. Kopra, V. Kotkas, M. Matskin and E Tyugu. "The NUT Language Report". Technical Report TRITA-IT-R 94:14, Teleinformatics/KTH, 1994.

[Uus95] T. Uustalu, “Aspects of Structural Synthesis of Programs”. Licentiate Thesis TRITA-IT-R 95:09, CSLab, Teleinformatics/KTH, 1995.

[VTA94a] V. Vlassov, E. Tyugu, M. Addibpour, “Distributed programming toolkit for NUT”. TRITA -IT R 94:34, CSLab, Teleinformatics/KTH, 1994.

[VKT93] B. Volozh, M. Köpp, E. Tyugu, “The NUT Graphics”. Technical Report TRITA-IT-R 93:05, CSLab, Teleinformatics/KTH, 1993.

[VTA94b] V. Vlassov, L-E. Thorelli, H. Ahmed, “Multi- EDA A Programming Environment for Parallel Computations”. Technical Report TRITA-IT-R 94:29, CSLab, Teleinformatics/KTH, 1994.

[YT89] A. Yonezawa, M. Tokoro (eds.), “Object-oriented concurrent programming”. The MIT Press, 1989.