KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology

Distributed Programming Toolkit for NUT

V. Vlassov, E. Tyugu, M. Addibpour

Department of Teleinformatics

KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology

Distributed Programming Toolkit for NUT

V. Vlassov, E. Tyugu, M. Addibpour

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

TRITA–IT R 94:34
ISSN 1103–534X
ISRN KTH/IT/R -- 94/34 -- SE

Department of Teleinformatics

Abstract

This document presents the Distributed Programming Toolkit rNUT users' guide and reference pages. The rNUT toolkit is an extension of the NUT System version 2.7 developed at KTH in June 1994. It consists of a library *librnut* and a daemon *nutd*. The library *librnut* contains low level NUT routines which provide support for building a system of collaborative NUT processes running on the Parallel Virtual Machine, PVM. To support interprocess communication, *librnut* contains routines for exchanging classes, scripts and objects between NUT processes. The daemon *nutd* is an executable code which is used to manage NUT process spawning and display connections. The synchronization and communication mechanism of rNUT is based on the Extended Dataflow Actor model, EDA, developed at KTH.

Contents

1 Introduction . . . 5
- 1.1 Scope and Purpose of this Document . . . 5
- 1.2 Other Documents on NUT . . . 5
- 1.3 Parallel and Distributed Computing in NUT . . . 5
  - 1.3.1 Control Structures for Parallel Computing . . . 5
  - 1.3.2 NUT Processes on PVM . . . 6
  - 1.3.3 Communication Operations . . . 7

2 PVM: The Parallel Virtual Machine . . . 7

3 The EDA Communication Operations . . . 7

4 Overview of rNUT . . . 8
- 4.1 Starting rNUT . . . 9
- 4.2 rNUT Processes . . . 9
- 4.3 Remote Process Control . . . 11
- 4.4 Passing of Classes . . . 11
- 4.5 Object Passing . . . 12
  - 4.5.1 Storing Routines . . . 12
  - 4.5.2 Fetching Routines . . . 12
  - 4.5.3 Mutually Exclusive Communication . . . 13
  - 4.5.4 Object Streams . . . 14
  - 4.5.5 Write-Once Shared Objects . . . 15
- 4.6 Passing of Workspace . . . 16
- 4.7 Perform Requests . . . 17

5 Reference Pages for rNUT Routines . . . 18

Conclusion . . . 45

Acknowledgements . . . 45

References . . . 45

Appendix A1: Grid and Mygrid Classes . . . 46
- A1.1 Class grid . . . 47
- A1.2 Class mygrid . . . 49

Appendix A2: A Parallel Search Problem . . . 52
- A2.1 Breadth-First Search . . . 52
- A2.2 Implementation . . . 53
- A2.3 Test Example . . . 55
- A2.4 Texts of Classes . . . 57

1 Introduction

1.1 Scope and Purpose of this Document

This document is an extension of the documentation of the NUT System version 2.7 developed at KTH in June 1994. It contains a user's guide and reference pages of functions needed for creating and running communicating NUT processes. These functions have been programmed in C language and included into the NUT library. The current version of NUT (version 2.7) extended with the distributed programming toolkit is called **rNUT** in the present paper.

The reader of this report is assumed to be acquainted with the principles of object-oriented programming and distributed computing as well as with the NUT system.

1.2 Other Documents on NUT

- T. Uustalu, U. Kopra, V. Kotkas, M. Matskin, E. Tyugu. *The NUT language report.* Technical Report TRITA-IT R 94:14. Dept. of Teleinformatics, KTH. June 1994. Available by anonymous ftp from **`it.kth.se`**, file **`Reports/TELEINFORMATICS/TRITA-IT-9414.ps.Z.`**

- Enn Tyugu. *The NUT system.* June 1994. Available by anonymous ftp from **`it.kth.se`**, file **`Software/CSlab/Software-Engineering/NUT/doc/syst.ps.Z`**.

- Benjamin Volozh, Mari Kõpp, Enn Tyugu. *The NUT Graphics*. Technical Report TRITA-IT-R 93:05, Dept. of Teleinformatics, The Royal Institute of Technology, June 1993. Available by anonymous ftp from **`it.kth.se`**, file **`Reports/TELEINFORMATICS/TRITA-IT-9305.ps.Z`**.

- Benjamin Volozh. *Appendix to The NUT Graphics.* March 1994. Available by anonymous ftp from **`it.kth.se`**, file **`Software/CSlab/Software-Engineering/NUT/doc/graphics-new.ps.Z`**.

- *The NUT libraries.* June 1994. Available by anonymous ftp from **`it.kth.se`**, file **`Software/CSlab/Software-Engineering/NUT/doc/libraries.ps.Z`**.

- *Interoperability of NUT with C and UNIX.* June 1994. Available by anonymous ftp from **`it.kth.se`**, file **`Software/CSlab/Software-Engineering/NUT/doc/interoperab.ps.Z`**.

- Bo Andersson, Benjamin Volozh. *The user interface of NUT.* June 94. Available by anonymous ftp from **`it.kth.se`**, file **`Software/CSlab/Software-Engineering/NUT/doc/userinterface.ps.Z`**.

1.3 Parallel and Distributed Computing in NUT

The functions described in the present report support coarse-grained (MIMD) distributed programming in NUT. These functions use the PVM distributed computing platform [5, 6] 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. Communication and synchronization between the NUT processes can be performed by sending store and fetch messages complying with the extended dataflow actor model (EDA) [3, 4]. A collection of functions has been developed also for passing classes, objects and scripts between the NUT processes. Besides the distributed computing, scalability of the software written in NUT is achieved by the usage of PVM, because spawning the NUT processes provides an unlimited object memory.

1.3.1 Control Structures for Parallel Computing

The NUT specification language enables one to describe computations in a compositional way, by combining computations described in subproblems of relations into larger programs [2]. A relation with subproblems can be considered as a control structure which composes a new algorithm from algorithms for solving subproblems. Using functions introduced in this report, 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 1.

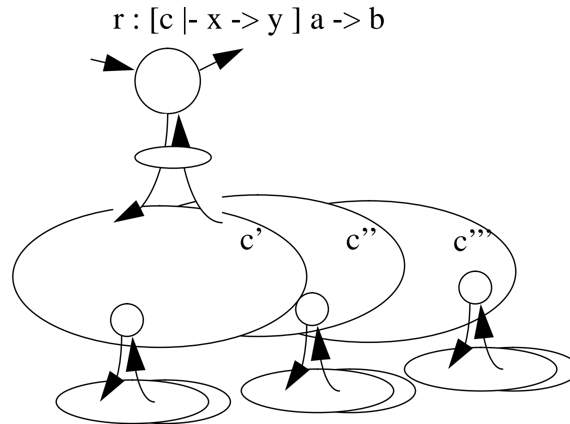

Figure 1. Relation r with subproblems controls computations on copies of a class

The relation **r** in Figure 1 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 1 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.

Examples of such structures were programmed by Urmas Kopra during his visit to KTH in spring 1994 [7]. He used the PVM directly for programming operators like *pararr* for parallel processing of elements of arrays.

1.3.2 NUT Processes on PVM

A distributed program developed by means of the present 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. The NUT package contains classes, objects and a script in the workspace [1]. 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 started first is called a *root process*. Any process can spawn, using one function call, a number of new processes which together with the process starting them constitutes a collection of processes called a grid. Figure 2 shows a snapshot of a dynamically changing pool of NUT processes (represented by rectangles) running on PVM in a network of different machines.

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.

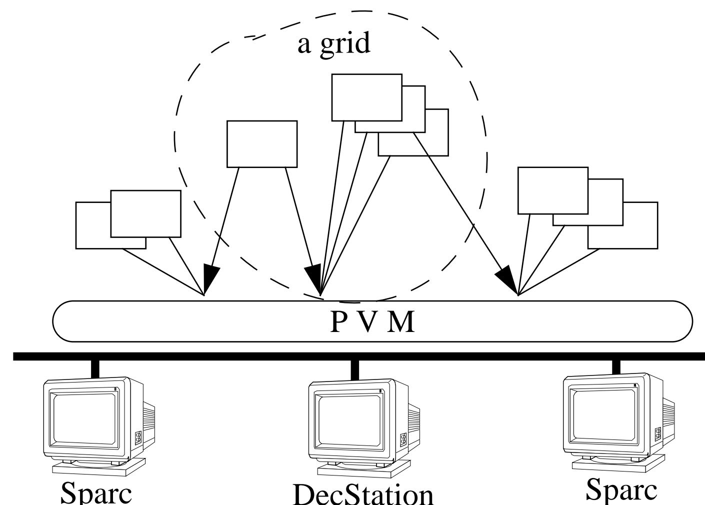

Figure 2. A tree of NUT processes

1.3.3 Communication Operations

In the present toolkit, functions are provided for the following kinds of operations

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

These operations are described in detail in the following chapters whereas the parallel virtual machine PVM is only briefly outlined and references are given to its documentation.

2 PVM: The Parallel Virtual Machine

PVM is an integrated framework for heterogeneous computing [5, 6]. A multitasked PVM program runs on a set of networked computers which are composed into a virtual multiprocessor. PVM supports parallel computation at the level of Unix processes, using message passing for communication.

PVM is composed of the PVM daemon, *pvmd*, and the PVM library, *libpvm*. An instance of the *pvmd* runs on each host of the virtual machine and provides host-process interactions. The *libpvm* forms an interface between PVM applications and the run-time system (PVM daemons).

Each task of a multitasked PVM application obtains a unique identifier, *tid*, which is used to specify the destination address for message passing. A PVM task can be spawned from another active task. PVM tasks communicate using tagged messages. A task can send messages asynchronously, and is responsible for receiving messages directed to it using blocking or non-blocking receive procedures.

The PVM library, *pvmlib,* includes the following groups of functions.

- Process control functions are used to start *pvmd*, to spawn new tasks, to enrol into PVM and to exit and halt PVM.
- Informative functions are used to get information about virtual machine configuration, running tasks and incoming messages.
- Configuration functions are used for adding hosts and deleting them from the virtual machine.
- Communication functions are used for packing data and sending messages, receiving messages and unpacking them.

A detail description of the PVM system can be found in [5].

3 The EDA Communication Operations

This section briefly and informally presents communication aspects of the Extended Dataflow Actor model, EDA, used for object passing in the rNUT system. A formal description and software implementation of the EDA model can be found in [3, 4].

EDA is a model of object-oriented multithreaded computation. 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. An EDA object can store the local data into shared memory and fetch a value of shared variables into its local memory. A shared variable may be in one of two states: full (containing data) or empty.

EDA defines the following operations on shared variables[1].

- *x-fetch* is a blocking extract operation. It is used for extracting the data from a shared variable to a local variable of an object. If the shared variable has the full state its value is extracted and its state becomes empty. If the shared variable is empty then *x-fetch* request is enqueued on this shared variable, and the executing thread is suspended until the variable becomes full by the matching *x-store*, *s-store*, or *i-store* operation.
- *i-fetch* is a blocking copy operation. It is used for copying the data from a shared variable to a local variable of an object. If the shared variable has the full state its value is copied and its state remains full. If the shared variable is empty then *i-fetch* request is enqueued on this shared variable, and the executing thread is suspended until the variable becomes full by the matching *x-store*, *s-store*, or *i-store* operation.

1. This list contains only those EDA operations which are used in the rNUT system.

● *x-store* is a blocking store operation. It is used for storing the data to a shared variable from a local variable of an object. If the shared variable is empty a local value is simply copied into the shared variable and its state becomes full. If the shared variable is full then *x-store* request is enqueued on this shared variable, and the executing thread is suspended until the variable is emptied by the matching *x-fetch* operation.

● *s-store* is a non-blocking buffering store operation. It is used for storing the data to a shared variable from a local variable of an object without suspension. If the shared variable is empty a local value is simply copied into the shared variable and its state becomes full. If the shared variable is full then computation in the executing thread resumes as soon as the local value is buffered. The buffer with the local data is enqueued on this shared variable until the variable is emptied by the matching *x-fetch* operation.

● *i-store* is a non-blocking store operation, which can be called as write-once. It is used for storing the data to a shared variable from a local variable of an object without suspension. If the shared variable is empty a local value is simply copied into the shared variable and its state becomes full. If the shared variable is full then *i-store* operation is ignored.

Defining shared memory operations, EDA recognizes three kinds of shared variables: *x, i* and *s*, each with special synchronization requirements [3]. Semantics of typed shared variable of EDA can be used in rNUT on the level of communication model and needs an extension in terms of operations to simplify rNUT application design and make it more convenient and flexible for NUT programming technology.

Table 1: Communication Operations

Name	Action	full	empty
x-fetch	extract a meaning value from a shared variable to local	extract	suspend
i-fetch	copy a meaning value from a shared variable to local	copy	suspend
s-fetch	extract any value from a shared variable to local	extract	return *empty*
u-fetch	copy any value from a shared variable to local	copy	return *empty*
x-store	store a local value to an empty shared variable	suspend	store
i-store	store a local value to an empty shared variable	skip	store
s-store	store a local value to an empty shared variable	buffering	store
u-store	update a local value to shared variable	update	store

Based on this argument, we specify the following additional communication operations, some of them have the same names as EDA operations, but have small differences in their semantics.

● *s-fetch* is a non-blocking extract operation. It is used for extracting the data from a share variable to a local variable of an object. If the shared variable has the full state its value is extracted and its state becomes empty. If the shared variable is empty then *s-fetch* extracts an empty value.

● *u-fetch* is a non-blocking copy operation. It is used for coping the data from a shared variable to a local variable of an object. If the shared variable has the full state its value is copied and its state remains full. If the shared variable is empty then *i-fetch* copies an empty value.

● *u-store* is a non-blocking unconditional update operation. It is used for updating a value of a share variable with a value local variable of an object. *u-store* always copies a local value into the shared variable independently of its state.

EDA operations and extension are summarized in Table 1.

4 Overview of rNUT

The toolkit rNUT consists of a library *librnut* and a daemon *nutd*. The synchronization and communication mechanism of rNUT is based on the Extended Dataflow Actor model, EDA [3]. *Librnut* is a library of low level NUT routines which provide support for building a system of collaborative NUT processes running on PVM [5]. To support interprocess communication, *librnut* contains routines for exchanging classes, workspaces and objects between NUT processes. The daemon *nutd* is an executable code which is used to manage NUT process spawning and display connections.

4.1 Starting rNUT

For some NUT applications, it is more convenient to define a configuration of PVM before starting rNUT. The easiest way to start PVM and define its configuration is to use the PVM console called *pvm* [5]. Two available console command *add* and *delete* are used for adding and deleting hosts of the virtual machine. The command *conf* lists the current configuration of the virtual machine. For example, a command:

```
pvm> add sole stinger bream walrus whale
```

- adds five hosts **sole, stinger,..., whale** to the virtual machine; The following command deletes two hosts from the virtual machine:

```
pvm> delete walrus sole
```

It is not necessary, however, to start PVM before starting the first NUT process called *root* NUT. The root NUT can be considered as an ordinary NUT process. Most of routines from *librnut* enrol the root NUT process into PVM. Each of them starts PVM if it is not running already, and spawns rNUT daemon *nutd* on the host which will be used to display NUT windows. *Nutd* is responsible for display connection control and for halting PVM when all NUT processes exit or are killed.

The library of the rNUT contains two routines *rnut_addhosts(nhosts, hosts)* and *rnut_delhosts(nhosts, hosts)* which are used to change the configuration of the virtual machine. For example, five hosts will be added by

```
rnut_addhosts(5, [sole, stinger, bream, walrus, whale]);
```

These routines also notify the daemon *nutd* about hosts which are added or deleted from PVM. *Nutd* correspondently adds or removes given hosts from the list of hosts allowed to connect to the X server to open a display connection.

4.2 rNUT Processes

The Distributed NUT System, rNUT, is structured as a number of collaborative NUT processes running on PVM. The NUT process starting first is called root NUT process. Any number of child NUT processes can be spawned from any active NUT using *rnut_spawn(nnut, mode, where, package)* routine (see Reference Pages). This routine starts up *nnut* NUT processes running in mode defined by *mode* on a host named *where* and loads a package named *package* into all spawned NUT processes. A child process can be spawned with open or closed window interface. If argument *where* is not defined or *nil* then PVM is responsible to choose the convenient set of hosts to start up new NUT processes. Each spawned child obtains a unique PVM task identifier, *tid*, which is used to specify the destination address for data passing and remote control. Each NUT process running in PVM can get its task identifier using *rnut_mytid()* routine, which returns its tid.

In the following example (see Figure 3) two NUT processes start with open window interface on the host called **'walrus.electrum.kth.se'** with package **'eda.mem'**. An array **tids** contains task identifiers of children.

Example 1.

```
mytid := rnut_mytid();
tids := rnut_spawn(2, 2, 'walrus.electrum.kth.se', 'eda.mem');
```

During spawning, the parent NUT sends the task identifiers of its children to the daemon *nutd* and also to each newly spawned child in order to notify them about their siblings. Each child can get this information if needed using *rnut_parent()* and *rnut_mygrid()* routines (see Figure 3). The first routine returns tid of the parent, the second one returns an array of tids of all siblings including the calling NUT process.

Example 2.

```
parent:= rnut_parent();
stids := rnut_mygrid();
```

A parent and all its child NUT processes spawned by the same action *rnut_spawn* compose a complete connected graph of NUT processes, which we call a *grid*. Spawn routine guarantees the same structure of copies of the array of tids which are distributed among all members of the grid. Each member of the grid can spawn their children and forms a new grid, being itself a parent. Figure 4 illustrates a process of spawning as a tree, where nodes are instances of NUT and arcs present spawn relations.

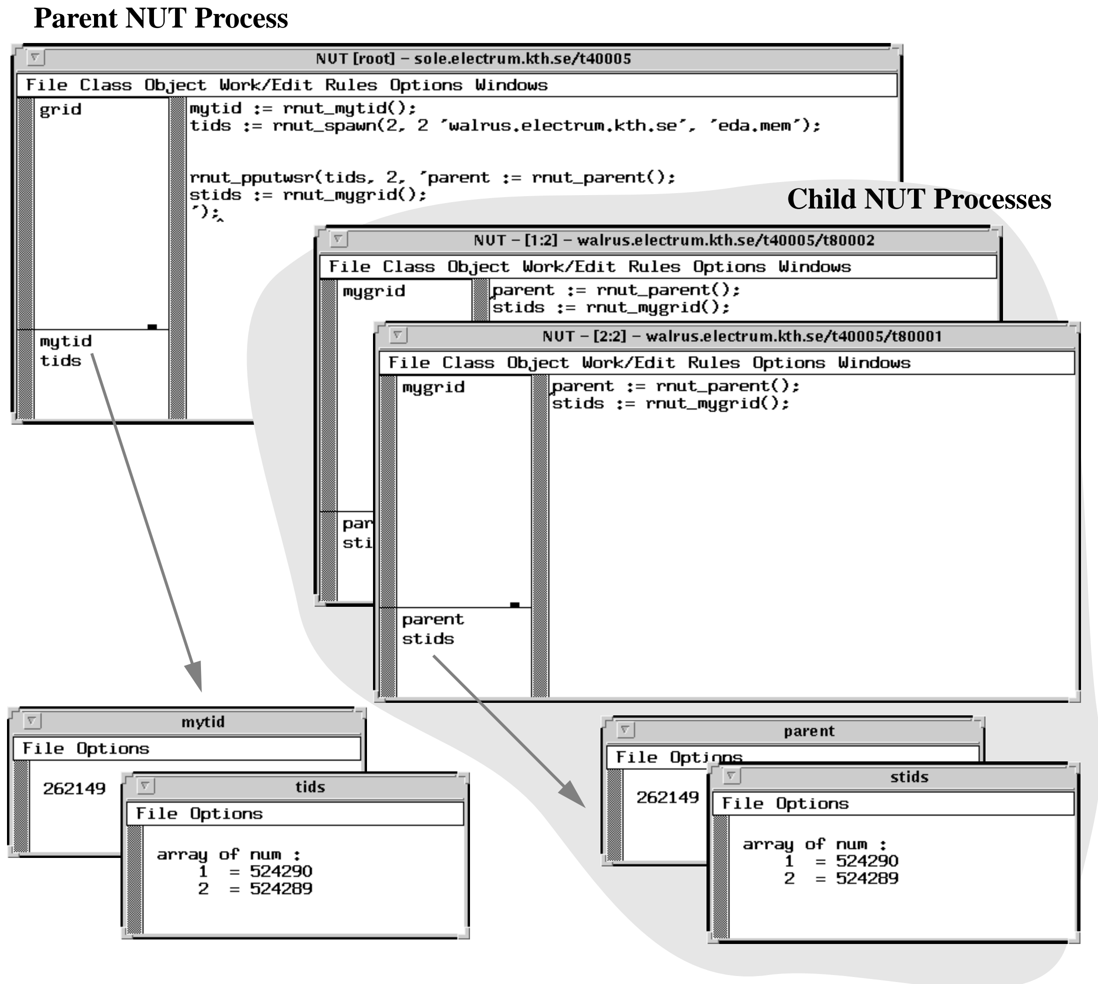

Figure 3. Parent and Child NUT Processes

Most of rNUT routines use tids as destination addresses for message passing. However, it is more convenient to use a metrics of integer numbers 0, 1, 2,..., n instead of task identifiers *mytid*, *tids* (Example 1) or *parent*, *stids* (Example 2) for addressing of NUT processes from the grid. If a rNUT routine is called from the parent NUT then number 0 is transformed into *mytid* and numbers 1, 2,..., n - into *tids[1], tids[2],..., tids[n]*, correspondently (see Example 1). If rNUT routine is called from a child NUT process then number 0 is transformed into *parent* and numbers 1, 2,..., n - into *stids[1], stids[2],..., stids[n]*, correspondently (see Example 2).

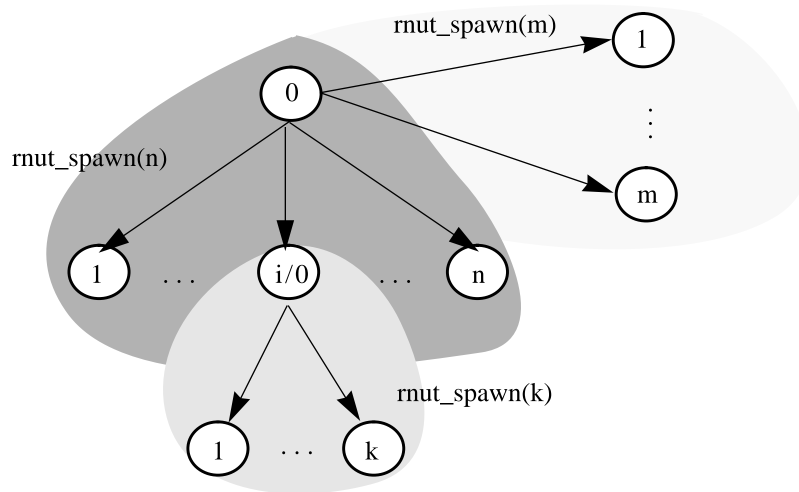

Figure 4. Spawn tree

Appendix 1 contains a specification of classes *grid* and *mygrid*, which are used to simplify the addressing scheme in rNUT routines. The first relation *gettids* converts array of numbers of members of the grid (called nodes) into array of tids. The *grid* class can be used in parent NUT process to specify a spawned grid as an object of *grid* class. The *mygrid* class is used in each child NUT process to access its sibling and parent using their numbers. The routine *rnut_mynum()* can be used to get the instance number of the calling NUT process in the group of its siblings, i.e. in the grid.

The PVM termination is realized using a semaphore in *nutd*. During spawning, the parent NUT notifies the daemon about its children and the semaphore is incremented by the number of spawned NUT processes. Each NUT process can exit independently of other processes. When a task exits or is killed, the PVM daemon *pvmd* notifies the rNUT daemon *nutd*, and the semaphore is decremented. When the semaphore becomes zero, *nutd* halts PVM.

The asynchronous exit of NUT processes can be the reason of deadlock while accessing shared objects, classes or workspaces. But interactive nature of the NUT programming technology, as well as remote process control routines give a possibility to handle these deadlocks and to solve the termination problem.

4.3 Remote Process Control

The library *librnut* includes routines for remote control of NUT processes (see Reference Pages). There are routines to close and to open a window interface of the remote NUT (*rnut_chmod*), to kill (*rnut_kill*) or to send an exit request to set of NUT processes (*rnut_exit*), to load a package (*rnut_pack*). The remote control of a NUT process from another NUT process is implemented by sending appropriate control requests, which can be consumed and served when the X-event loop in the receiving NUT process is idle and the process is not in the Intepreter stage.

A service of the exit request generated with *rnut_exit* routine and the package load request generated with *rnut_pack* routine notifies a user and asks a permission to exit or to change a package.

All control actions are performed asychronously, i.e without a feed back. It is reasonable, because a control request can not interrupt the NUT process, to which this request is directed, and a delay between sending the request and receiving a reply can be quite big.

4.4 Passing of Classes

The library *librnut* contains routines that enable a NUT process to send a copy of a text of a user-defined class to a set of NUT processes (*rnut_clcpy*), and to get a copy of a text of user defined class from another NUT process (*rnut_clget*). Both routines allow to rename the copy of the class in the destination NUT processes. In the following example, the first statement sends a copy of the class named *'grid'* with a new name *'gr'* to processes specified by an array *tids.* The next statement sends a request to get a copy of the class named *'mygrid'* with the same name *'mygrid'* from parent process:

```
rnut_clcpy(tids, length(tids), 'gr', 'grid');
rnut_clget(rnut_parent(), 'mygrid, 'mygrid');
```

Being received the text of the class is automatically compiled. If a NUT process already contains a class named as the received class then the new class overwrites the old one.

The parse algorithm of NUT Interpreter assumes that all statements, which are chosen from a workspace to be performed, are parsed as one unit. That is why the Interpreter notifies about an error when *rnut_clget* routine and objects of a class which is needed to be got are used in one unit chose from the workspace to be performed. For example when parsing the following set of statements as one unit the Interpreter will notify that class *mygrid* is empty and can not be used:

```
% first part
rnut_clget(rnut_parent(), 'mygrid', 'mygrid');
% second part
myg := new mygrid;
me := myg.mynum;
myg.ustore([me - 1, me + 1], 2, 'pi', 3.1415926);
```

To avoid this conflict the user can perform the given statements in two steps: first perform *rnut_clget* routine and then perform the second part, as shown with comments.

A user defined class is sent as a text which is compiled by the receiving process. If this process has not got some user-defined subclass of the received class, then NUT interpreter creates an empty class. To avoid this effect, it is important to send user-defined classes and subclasses in a correct order. It is safer and more convenient, however, to use package passing (see *rnut_pack*) instead of passing classes separately.

4.5 Object Passing

The library *librnut* provides communication routines for object passing based on the parallel model, called EDA (see Chapter 3 for further information).

The *librnut* contains eight routines of a general-purpose communication and synchronisation package. These are storing and fetching routines, which can be used in matching pairs:

- *rnut_xstore* and *rnut_xfetch* are blocking communication routines supporting mutually exclusive interprocess communication.
- *rnut_sstore* and *rnut_sfetch* are non-blocking communication routines supporting object streams between NUT processes.
- *rnut_istore* and *rnut_ifetch* are non-blocking communication routine supporting write-once shared object for synchronizing single writer-multiple readers.
- *rnut_ustore* and *rnut_ufetch* are non-blocking communication routine for unconditional updating of shared objects.

The EDA communication model assumes that communication operations are realized on a space of shared and local objects located in particular NUT processes. A shared (remote) object can be in one of two states: full, i.e. has a value, or empty, i.e. has the value *nil* or does not exist. The remote object is addressed by two components *tid* and *objname*:

- *tid* is a task identifier of a NUT process in which given object is located,
- *objname* is a name of the object.

A storing routine multicasts a request to store a value of a local object into shared objects located in remote NUT processes. A fetching routine sends a request to extract or to copy a value from a remote object into a local one. All fetching routines, as well as *rnut_xstore* routine, block the requesting process until a value or acknowledgments arrives. A requested remote process can serve the request only if it is not in the Interpreter stage, i.e. it does not perform anything.

4.5.1 Storing Routines

Storing routines have the following synopsis:

```
rnut_*store(tids, nnut, objname, obj)
```

Where the asterisk '`*`' should be one of four characters: **`x`**, **`s`**, **`i`**, or **`u`**.

Storing routines, *rnut_*store*, can be used to store a value of the object *obj* into shared objects, named *objname*, of the NUT processes, specified by array *tids*. A storing routine sends a value *obj* to NUT processes defined by *tids* with a request to store a value of *obj* into shared objects named *objname*. Three routines, *rnut_istore*, *rnut_sstore* and *rnut_ustore* are non-blocking. It means that computation on the requesting process resumes as soon as all requests are sent. The routine *rnut_xstore* blocks the requesting process until *nnut* acknowledgments have arrived from requested processes. The received value can be consumed in the receiving process by different ways according to an access type: *x*, *i*, *s*, or *u*, and state of the object *objname*. To provide an arbitrary access to shared objects rNUT uses *nil* value to define an empty state of an object.

4.5.2 Fetching Routines

Fetching routines have the following synopsis:

```
obj := rnut_*fetch(tid, objname);
```

Where the asterisk '`*`' should be one of four characters: **`x`**, **`s`**, **`i`**, or **`u`**.

Fetching routines, *rnut_*fetch*, can be used to extract or to copy a value into the local object *obj* from the shared objects, named *objname* and located in the NUT process, specified by *tid*. A fetching routine sends the name *objname* to NUT process defined by *tid* with a request to extract or to copy a value from an object, named *objname*. Two routines *rnut_xfetch* and *rnut_sfetch* generates an extract request, two other routines *rnut_ifetch* and *rnut_ufetch* generates an copy request. A requesting process becomes suspended until a value has arrived from a requested process. The received value is assigned to the local object *obj*.

Two routines, *rnut_sfetch* and *rnut_ufetch,* are non-blocking. It means that the requested process sends back *nil*, if the *objname* is empty, i.e. has the value *nil* or does not exist. Otherwise the requested process sends a value of the object named *objname* and resets this object to *nil*, in case of extract request. Once *nil* or the value has arrived, the fetch routine

pushes it onto the NUT stack, and the requesting process resumes. Moreover, if the requesting process does not have the class of the object named *objname*, then it gets this class from the requested process and compiles it.

Two other fetching routines, *rnut_xfetch* and *rnut_ifetch,* are blocking. It means that the requested process can execute fetch request only if the object named *objname* is full, i.e. has a value. Otherwise the fetch request is queued until the *objname* becomes full by the matching storing routine. Once a value of *objname* has been sent, the requested process resets this object to *nil*, in case of extract request. Once the value arrives, the requesting process pushes it onto the NUT stack and resumes.

4.5.3 Mutually Exclusive Communication

The EDA model provides *x*-operations, for accessing critical regions in mutual exclusion and supporting synchronous producer-consumer relationships [3]. To realise mutually exclusive communication through the space of shared objects the rNUT library involves a pair object passing routines *rnut_xstore* and *rnut_xfetch*. The storing routine *rnut_xstore* can succeed if a shared object is empty. Otherwise the store request is queued until the shared object is emptied by an extract request. The fetching routine *rnut_xfetch* extracts a value from a full shared object. If the shared object is empty then the extract request is queued until the shared object becomes full by a store request.

Another combination of object passing routines, *rnut_xstore* and *rnut_sfetch*, can be used to support the mutual exclusion relationship in more flexible way. The *rnut_sfetch* routine is non-blocking. It extracts a *nil* value form an empty object and a meaning value from a full object.

A NUT process performing the *rnut_xstore* routine sends data with *x*-store request, as mentioned in 4.5.1. Then the sending process becomes suspended until acknowledgments have arrived from all requested processes. At this stage, however, the process may serve any remote requests and replies excluding perform requests, which are buffered at the PVM level. To count acknowledgments, the requesting process uses a semaphore which is incremented by the number of requests. When an acknowledgment arrives, the semaphore is decremented. When it becomes zero, computation on the sending process resumes.

A receiving process can store a value and send an acknowledgement back only if the object named *objname* is empty, i.e. has *nil* value or does not exist. In the last case, the receiving process creates an object named *objname* from the same class as *obj* and stores the value. Moreover, if it does not have that class, then it gets the class including all subclasses from the requested process. The consuming of the value is visible in the same manner as creation of an object *objname* by means of *new* expression. After that a user can use the object *objname* in an ordinary way. However, the user is responsible for avoiding possible deadlocks, because if the object named *objname* is not empty, then the store request is queued until the *objname* will be emptied by some fetching routine.

To provide a correct synchronization of processes, and to avoid possible deadlocks, it is more convenient to consume values stored into shared objects by extracting them into local objects using matching fetching procedures: *rnut_xfetch* or *rnut_sfetch*. Each of these routines extracts a value from a full shared object and serves the first store request suspended in a store request queue of this object. If the waiting request is *x*-store then fetching procedure sends an acknowledgement to the requesting NUT process.

Figure 5 illustrates an example of using the pair of *rnut_xstore* and *rnut_xfetch* routines. A consumer process running on a host *sole.electrum.kth.se* stores a value of its object *x* into objects named '*shrd_x*' of two consuming processes specified by *tids*. Each consumer uses the *rnut_xfetch* to extract the value from shared object named '*shrd_x*' into its object *myx*.

The producing process sends a value into shared objects located in child processes:

```
rnut_xstore(tids, length(tids), 'shrd_x', x);
```

Each child process consumes data from its shared object:

```
myx := rnut_xfetch(rnut_mytid(), 'shrd_x');
```

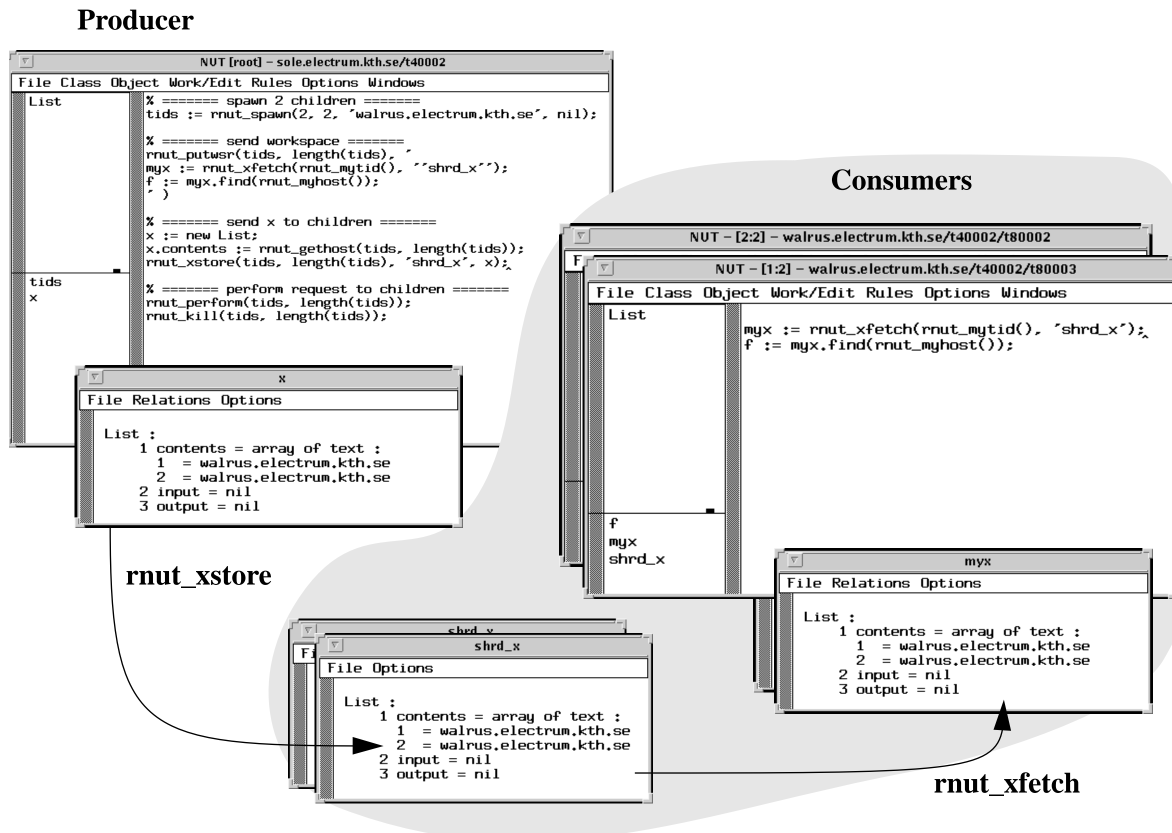

Figure 5. Object Passing (rnut_xstore, rnut_xfetch)

4.5.4 Object Streams

A pairs of matching routines *rnut_sstore* and *rnut_sfetch* (or *rnut_sstore* and *rnut_xfetch*) can be used to support an object stream between NUT processes. The *rnut_sstore* routine is non-blocking communication routine. As soon as the *s*-store request and data have been mulicasted, the routine returns.

A receiving process can serve the *s*-store request if the object named *objname* is empty, otherwise the request is queued until the objname becomes emptied by the matching *rnut_xfetch* or *rnut_sfetch* routine.

The following example illustrates using of *rnut_sstore* routine for sending a set of objects through the same shared object located in a remote process. The process *A* sends five objects of different classes into the shared object named '*sh_buf*' located in the remote NUT process specified by *tid_b*. The receiving process *B* extracts this data from the object *sh_buf* using *rnut_xfetch* routine.

Process *A*:

```
dest := [tid_b];
buf := 'sh_buf';
rnut_sstore(dest, 1, buf, 5);
rnut_sstore(dest, 1, buf, -1.3 );
rnut_sstore(dest, 1, buf, 'Hello World');
rnut_sstore(dest, 1, buf, [7, 3.1415]);
rnut_sstore(dest, 1, buf, ['radio', '-ga', '-ga']);
```

Process *B*:

```
x := new array of any;
me:= rnut_mytid();
for i := 1 to 5 do
x[i]:= rnut_xfetch(me, 'sh_buf');
od;
```

Figure 6 illustrates changing of a value of the object named '*sh_buf*' and the result value of the local object *x* which consumes a stream of values from the process A.

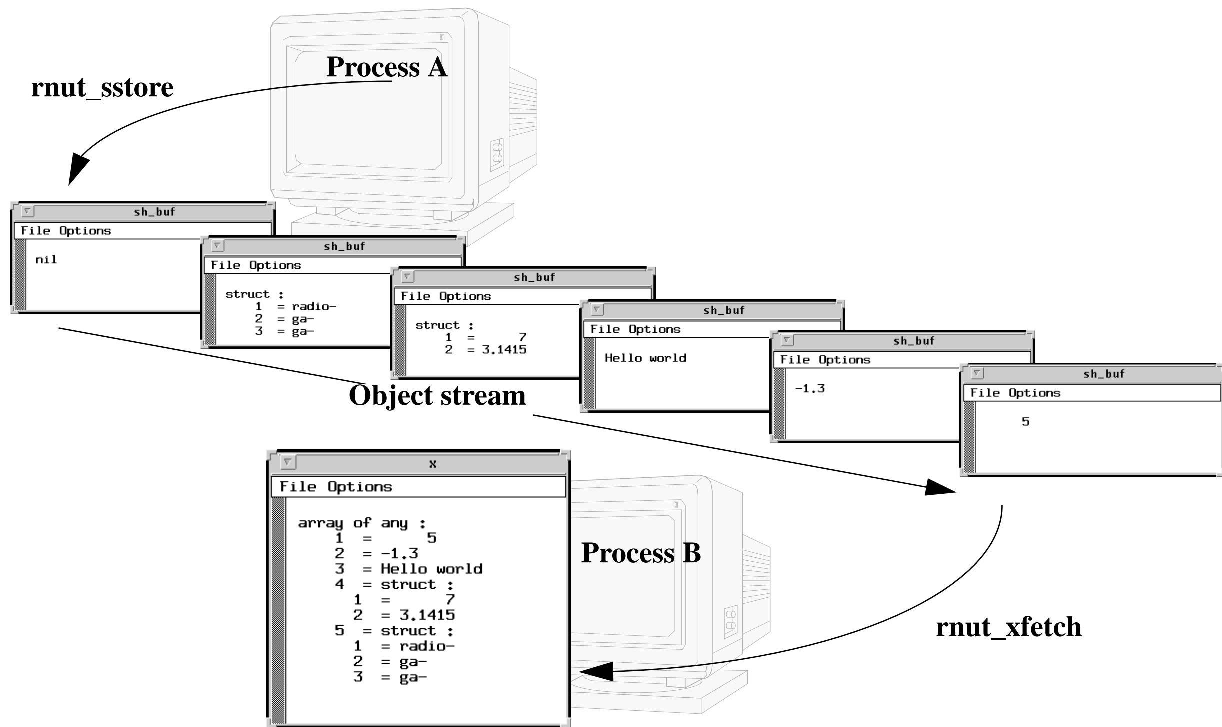

Figure 6. Stream of Objects (rnut_sstore, rnut_xfetch)

4.5.5 Write-Once Shared Objects

A pair of routines *rnut_istore* and *rnut_ifetch* (or *rnut_istore* and *rnut_ufetch*) supports write-once shared objects which can be used for synchronizing specific parallel computation schemes, such as single writer-multiple readers and a OR-parallelism [3].

The storing routine *rnut_istore* is a non-blocking routine, which returns as soon as a store request and a value are multi-casted to requested processes. A receiving process can store the value if the requested shared object is empty, i.e. has a *nil* value or does not exist. Otherwise, the store request is discarded. If several processes try to store values into the same shared object using *rnut_istore* routine, then only one of candidates succeeds.

The fetching routine rnut_ifetch is blocking routine, which generates a request to copy a value from a shared object to a local one. The request is satisfied if the shared object is full, otherwise it is queued until the shared object becomes full by a matching store routine.

The following example illustrates using of the pair *rnut_istore* and *rnut_ifetch* routines to realise the OR-parallelism. The process *Boss* chooses the first object from several objects sent to it by its child processes $Worker_1$,..., $Worker_N$. Each child process, $Worker_I$ (I = 1,..., N), sends a value of its local object *work_out* into the shared object named '*boss_in'* located in the process *Boss* specified by a task identifier *tid_b*. The receiving process *B* chooses one of several candidates as winner consuming its data from the object '*boss_in'* using *rnut_ifetch* routine (see Figure 7).

Process $Worker_I$ (I = 1,..., N):

```
tid_b := rnut_parent();
% compute worker_out
worker_out := rnut_mytid();
% send the value of worker_out to the Boss process
rnut_istore([tid_b], 1, 'boss_in', worker_out);
```

Process *Boss*:

```
. . .
% consume the f irst answer from children
my_in := rnut_ifetch(rnut_mytid(), 'boss_in');
```

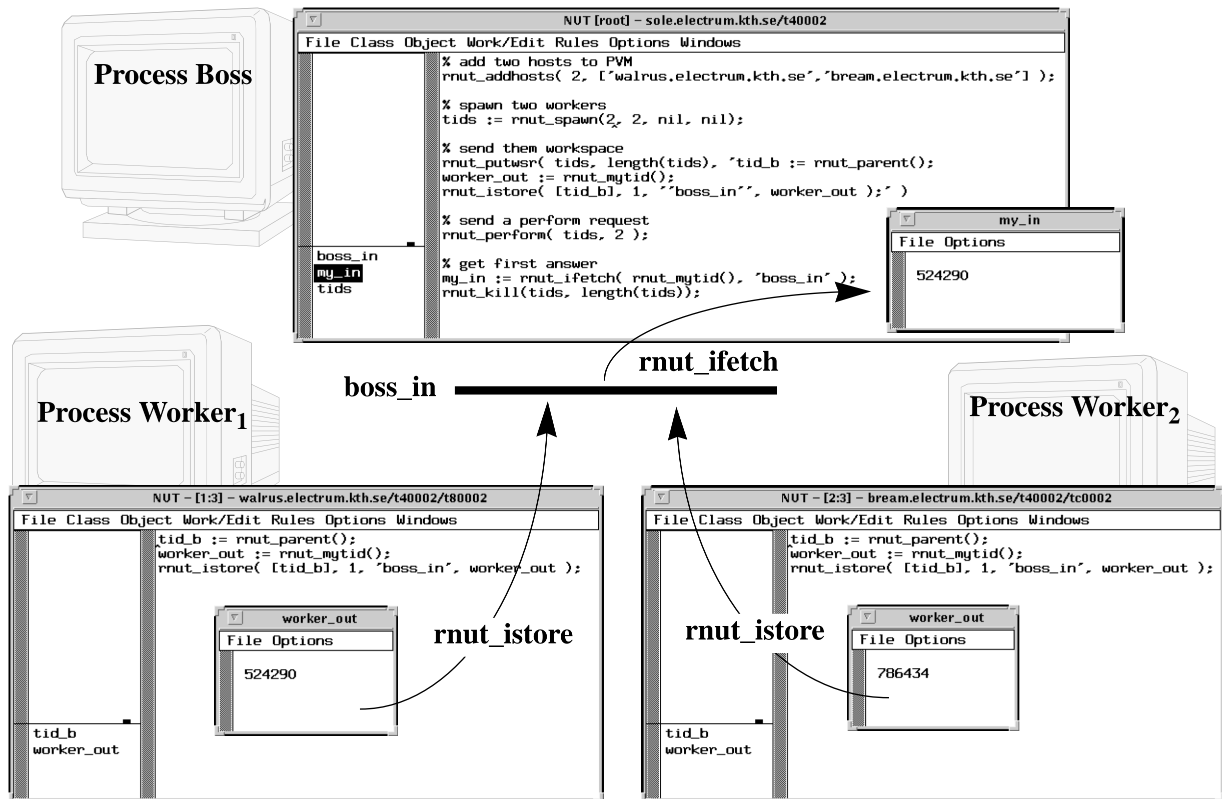

Figure 7. Write-Once Shared Object (rnut_istore, rnut_ifetch)

4.6 Passing of Workspace

The library *librnut* contains routines for passing of a text for the workspace:

```
rnut_putws*( tids, nnut, ws );
rnut_getws*( tid );
```

Where the asterisk '*' should be one of two characters: **a** (append) or **r** (replace).

Routines *rnut_putws** allow to multicast a text of workspace *ws* to *nnut* processes specified by *tids*. Routines *rnut_getws** are used to get a text of current workspace from remote NUT process specified by *tid*.

The received text is appended to the current workspace of the receiving process, if a routine has a suffix **a**. The received text replaces the current worspace of the receiving process, if a routine has a suffix **r** (see Table 2). In the following example a parent process multicasts the workspace to its children spawned by *rnut_spawn* routine.

Parent process:

```
% spawn 4 processes with open windows somewhere and send them a workspace
tids = rnut_spawn( 4, 2, nil, nil );
rnut_putwsr( tids, 4, '
      mygr := new mygrid;
      in := mygr.ifetch(0, ''args'');
');
% send class 'mygrid', input arguments and a perform request
rnut_clcpy(tids, 4, 'mygrid', 'mygrid');
rnut_istore(tids, 4, 'args', args);
rnut_perform(tids, 4);
```

Received work space of a child process:

```
mygr := new mygrid;
in := mygr.ifetch(0, 'args');
```

4.7 Perform Requests

The routine *rnut_perform* multicasts a request to perform a current workspace of requested processes. The library *librnut* also supplies the routines which combine passing of a workspace and a perform request into a single action. The routines *rnut_pputws**, *rnut_pgetws** are functions to allow the user to send or get a workspace with a request to perform it. These routines have the following synopsis:

```
rnut_pputws*( tids, nnut, ws );
rnut_pgetws*( tids );
rnut_perform ( tids, nnut );
```

Where the asterisk '**`*`**' should be one of two characters: **`a`** (append) or **`r`** (replace).

The routines *rnut_pputws** multicasts a text of a workspace *ws* to *nnut* remote NUT processes specified by *tids* with a request to perform it. The routines *rnut_pgetws** are used to get a text of workspace from a process *tid* and then perform it. The routine *rnut_perform* multicasts a perform request to *nnut* processes defined by *tids* (see Example in 4.6). If a requested process is not suspended then it performs its current workspace.

The user is responsible to provide in advance all data: objects and classes, which are necessary to perform the passed workspace.

All routines, excluding *rnut_perform*, which are used for passing workspaces and a perform requests are summarized in Table 2.

Tabell 2: Routines for passing of Workspaces and Perform Requests

	with perform request		without perform request	
put workspace	rnut_pputwsr	rnut_pputwsa	rnut_putwsa	rnut_putwsr
get workspace	rnut_pgetwsr	rnut_pgetwsa	rnut_getwsa	rnut_getwsr
	replace	append		replace

5 Reference Pages for rNUT Routines

This chapter contains reference pages for all rNUT routines used in the Distributed NUT Environment. Rounites are listed in an alphabetical oder. Each reference page contains simple examples.

rnut_addhosts, rnut_delhosts

Change the configuration of the virtual machine(3PVM).

Synopsis

```
info := rnut_addhosts( nhost, hosts )
info := rnut_delhosts( nhost, hosts )
```

Arguments

nhost Integer numeric object specifying the number of hosts to be added/deleted.
hosts Text object array of length ***nhost*** containing the names of the machines to be added/deleted.

Return values

info Integer numeric object returning the actual number of hosts added/deleted. Values less than ***nhost*** indicate partial failure, and values equal ***nil*** indicate total failure.

Description

Routines ***rnut_addhosts***, ***rnut_delhosts*** add/delete the computers named in hosts changing the configuration of the PVM(3PVM). ***hosts*** should be an array of host names such as [‘sole.electrum.kth.se’, ‘bream.electurm.kth.se’]. Routines use the ***pvm_addhosts***(3PVM) and ***pvm_delhosts***(3PVM) functions of the pvm library to add or delete hosts.

The routine ***rnut_addhosts*** called from NUT process, running on the host (X host), used for display the NUT window interface, adds given hosts to the list of hosts allowed to connect to the X server to open a display connection. The routine ***rnut_delhosts*** called from NUT process, running on X host, removes given hosts from the list of hosts allowed to connect to the X server. If ***rnut_addhosts*** or ***rnut_delhosts*** is called from NUT process, running not on the X host, then the routine sends given host names to the NUT daemon nutd running on the X host. ***nutd*** adds or deletes host names from the X access list, using ***xhost***(1).

Examples

```
list := [‘whale’,’walrus’,’cod’];
n := rnut_addhosts( length(list), list );
info := rnut_delhosts( 1, ‘bream.electrum.kth.se’ );
```

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***.

rnut_chmod

Change a mode of NUT processes.

Synopsis

```
info := rnut_chmod( tids, nnut, mode )
```

Arguments

tids Numeric array of length ***nnut*** containing tids of destination NUT processes. It can contain a tid of the calling NUT.

nnut Integer numeric object specifying a number of NUT processes to change the mode.

mode Integer numeric object specifying an user interaction mode to set for destination NUT processes. The parameter ***mode*** should have one of values:

Mode value	MEANING
1	NUT outside the PVM
2	NUT with open window interface (default)
3	NUT with closed window interface

Return values

info The routine returns the numeric ***mode*** in ***info***. On failure, it returns ***nil***.

Description

The routine ***rnut_chmod*** sends a value of a mode ***mode*** to ***nnut*** NUT processes specified by ***tids*** with a request to set this mode. Each receiving NUT sets this mode. This routine can be used to close or open window interface of collob-orative NUT processes running in the distributed NUT environment. If mode = 1 then the receiving NUT process exits PVM, and can not be used for object passing.

Examples

```
tids := rnut_spawn(3, 2);
. . % prepare and send a work for children tids
rnut_chmod( tids, length(tids), 3);
. . % get a result from children tids
rnut_chmod( tids, length(tids), 2);
```

Errors

rnut_chmod returns ***nil*** in error situations. In the open user interaction mode, on failure, ***rnut_chmod*** displays a win-dow to notify the user about the error. In the closed user interaction mode ***rnut_chmod*** prints an error message to ***stderr***.

rnut_clcpy

Mulicasts a class copy to remote NUT processes.

Synopsis

```
info := rnut_clcpy( tids, nnut, rclname, lclname )
```

Arguments

tids	Numeric array of length ***nnut*** containing tids of destination NUT processes. It can contain a tid of the calling NUT.
nnut	Integer numeric object specifying a number of NUT processes to copy a class.
rclname	Text object specifying a new name of the copied class in the destination NUT processes.
lclname	Text object specifying a name of the class to copy.

Return values

info	The routine returns the name ***lclname*** in ***info***. On failure, it returns ***nil***.

Description

The routine ***rnut_clcpy*** is used to copy a content of the class named ***lclname*** located in the calling NUT into classes named ***rclname*** located in remote NUT processes. The routine sends a name **rclname** with a text of the class ***lclname*** to ***nnut*** NUT processes identified by ***tids.*** Each receiving NUT creates the class named ***rclname*** with the text of class ***lclname*** and compiles it. If receiving NUT already contains the class named ***rclname*** its old contens is replaced by received new contents.

Example

```
tids:= rnut_spawn(3, 2);
rnut_clcpy( tids, length(tids), 'grid', 'ch_grid');
```

Errors

rnut_clcpy returns ***nil*** in error situations. In the open user interaction mode, on failure, ***rnut_clcpy*** displays a window to notify the user about the error. In the closed user interaction mode ***rnut_clcpy*** prints an error message to ***stderr***.

rnut_clget

Gets a copy of a class from remote NUT process and compiles it.

Synopsis

```
info := rnut_clget( tid, rclname, lclname )
```

Arguments

`tid`	Integer numeric objec specifying a tid of the remote NUT process.
`rclname`	Text object specifying a name of the class to get from the remote NUT .
`lclname`	Text object specifying a new name of the class in the calling NUT process.

Return values

`info`	The routine returns the name ***rclname*** in ***info***. On failure, it returns ***nil***.

Description

The routine ***rnut_clget*** is used to get a copy of the class named ***rclname*** located in the remote NUT specified by ***tid*** into class named ***lclname*** located in the calling NUT. The routine sends a name ***rclname*** to NUT identified by ***tid*** with a request to send back a text of the class ***rclname .***

The routine ***rnut_clget*** is asynchronous, i.e. computation in the calling NUT continues as soon as the request to get a class is sent to the remote NUT. It is necessary to prevent a dealock if the remote NUT does not jet have the class ***rclname***. Whenever the text of the class ***rclname*** is arrived the calling NUT compiles it with the name ***lclname***. This rotine can not be used for data synchronization.

Example

```
rnut_clget( rnut_parent(), 'mygrid', 'mygrid');
```

Errors

rnut_clget returns ***nil*** in error situations. In the open user interaction mode, on failure, ***rnut_clget*** displays a window to notify the user about the error. In the closed user interaction mode ***rnut_clget*** prints an error message to ***stderr***.

rnut_exit

Multicasts exit requests to remote NUT processes.

Synopsis

```
rnut_exit( tids, nnut )
```

Arguments

tids Numeric array of length ***nnut*** containing tids of destination NUT processes. It can contain a tid of the calling NUT.

nnut Integer numeric object specifying a number of NUT processes to send exit requests.

Description

The routine ***rnut_exit*** is used to multicast an exit request to each process from a set of remote NUT processes identified by ***tids***. Each destination NUT tries to exit, i.e. if it has changed open package it displays a window asking a permission to exit.

When each of running NUT processes exits or is killed the NUT daemon ***nutd*** deletes all hosts added to a configuration of PVM from a list of hosts allowed to connect to the root X server. Then ***nutd*** halts PVM.

Example

```
tids:= rnut_spawn(3, 2);
..
rnut_exit( tids, length(tids));
```

Errors

In the open user interaction mode, on failure (e.g. bad parameters), ***rnut_exit*** displays a window to notify the user about the error. In the closed user interaction mode ***rnut_exit*** prints an error message to ***stderr***.

rnut_*fetch

Fetch an object from a NUT process.

Synopsis

```
obj := rnut_xfetch( tid, objname )
obj := rnut_sfetch( tid, objname )
obj := rnut_ifetch( tid, objname )
obj := rnut_ufetch( tid, objname )
```

Arguments

`tid`	Numeric task identifier of the requested NUT process.
`objname`	Text object containing the name of the object to be fetched.

Return values

`obj`	Each of the fetch routines returns the object ***obj*** that is a value of the object named ***objname*** in the requested NUT process. On failure, they return ***nil***.

Description

These are fetching routines of a general-purpose communication and synchronization package, based on the parallel model, called ***EDA***, developed at the Department of Teleinformatics of the Royal Institute of Technology, Sweden []. They can be used to fetch a value of the object, named ***objname***, from NUT process, specified by ***tid*** into an object ***obj***. This NUT process can be spawned by ***rnut_spawn***(3NUT).

rnut_xfetch is a blocking communication routine supporting mutually exclusive interprocess communication, as well as object streams between NUT processes. It sends a request to the NUT process specified by ***tid*** to extract a value from an object named ***objname***. The routine blocks the calling NUT process until a value arrives. NUT process receiving the request sends a reply with value as soon as extract operation has been performed. The requested process can execute extract request only if the object named ***objname*** is full, i.e. has a value. Otherwise the fetch request is queued until the ***objname*** becomes full by the matching ***rnut_xstore***(3NUT), ***rnut_sstore***(3NUT), ***rnut_istore***(3NUT) or ***rnut_ustore***(3NUT). Moreover, if the requesting process does not have the class of the object named ***objname***, then it gets this class from the requested process and compiles it. Once a value of ***objname*** has been sent, the requested process resets this object to nil. Once the value arrives, the requesting process pushes it onto the NUT stack and returns from ***rnut_xfetch***.

rnut_sfetch is a non-blocking communication routine supporting mutually exclusive interprocess communication as well as object streams between NUT processes. It sends a request to extract a value from an object named ***objname*** of NUT process specified by ***tid***, onto the NUT stack of the requesting NUT process, i.e. into an object ***obj***. NUT process receiving the request sends back ***nil***, if the ***objname*** is empty, i.e. has the value ***nil*** or does not exist. Otherwise the process ***tid*** sends a value of the object named ***objname*** and resets this object to ***nil***. Once ***nil*** or the value have arrived, ***rnut_sfetch*** pushes it onto the NUT stack and returns.

rnut_ifetch is blocking routine supporting multiple-read from shared objects. It sends a request to copy a value from an object named ***objname*** of NUT process specified by ***tid*** onto the NUT stack of the requesting NUT process, i.e. into an object ***obj***. The routine blocks the requesting NUT process until a value have arrived from the requested process. NUT process receiving the request sends a reply with value as soon as fetch operation can be performed. This can be done only if the object named ***objname*** is full, i.e. has a value. Otherwise the copy request is queued until the ***objname*** becomes full by the matching ***rnut_xstore***(3NUT), ***rnut_sstore***(3NUT), ***rnut_istore***(3NUT) or ***rnut_ustore***(3NUT). If the requesting process does not have the class of the object named objname, then it gets this class from the requested process and compiles it. Once a value arrives, the requesting process pushes it onto the NUT stack and returns from ***rnut_ifetch***. The difference between ***rnut_xfetch*** and ***rnut_ifetch*** is that the last one makes copy of the object named ***objname*** and does not reset it to ***nil***, as in the case of ***rnut_xfetch***.

rnut_ufetch is non-blocking communication routine supporting multiple-read from shared objects. It sends a request to copy a value from an object named ***objname*** of NUT process specified by ***tid*** onto the NUT stack of the requesting NUT process, i.e. into an object ***obj***. NUT process receiving the request sends back ***nil***, if the object named ***objname*** is empty, i.e. has the value ***nil*** or does not exist. Otherwise it sends a copy of the value of ***objname***. Once ***nil*** or the value arrives, ***rnut_sfetch*** pushes it onto the NUT stack and returns.

Examples

```
p := rnut_parent();
my_a := rnut_xfetch(p, 'a');
. .
probe := rnut_sfetch(tid, 'stream');
if probe /= nil ->
     % use 'probe' with new value
     . .
true ->
     % go do other computing
     . .
fi;
. .
b := new myclass;
b.input := [-1.3, 'minus one point three'];
rnut_ufetch(tids, n, 'ss', b);
```

Errors

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

rnut_gethost

Gets host names of specified NUT processes.

Synopsis

```
hosts:= rnut_gethost( tids, nnut )
```

Arguments

tids Numeric array of length ***nnut*** containing the tids of NUT processes in question. It can contain the tid of the calling NUT.

nnut Integer numeric object specifying the number of Nut processes in question.

Return values

hosts Text object array of length ***nnut*** containing the names of the machines running NUT processes specified by ***tids***.

Description

The routine ***rnut_gethost*** returns a list of host names on wich NUT processes identified by ***tids*** are located.

Example

```
tids:= rnut_spawn( 5, 3, nil, nil);
hosts:= rnut_gethost(tids, length(tids));
```

Errors

rnut_gethost returns ***nil*** in error situations. In the open user interaction mode, on failure, ***rnut_gethost*** displays a window to notify the user about the error. In the closed user interaction mode ***rnut_gethost*** prints an error message to ***stderr***.

rnut_getwsa, rnut_getwsr

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

Synopsis

```
rnut_getwsa( tid )
rnut_getwsr( tid )
```

Arguments

tid Integer numeric objec specifying a tid of the remote NUT process.

Description

Routine ***rnut_getwsa*** and ***rnut_getwsr*** are used to get a copy of the text of the Workspace from the remote NUT specified by ***tid*** and to place it into the Workspace of the requesting NUT. The routines send a request to get back a text of the Workspace from the remote NUT.

Routines are asynchronous, i.e. computation in the requesting NUT continues as soon as the request to get the Workspace is sent to the remote NUT. Whenever the text of the Workspace is arrived the requesting NUT appends it to its current Workspace in the case of ***rnut_getwsa***. In the case of ***rnut_getwsr*** the new Workspace from remote NUT replaces the current Workspace of the requesting NUT.

Example

```
rnut_getwsa( rnut_parent() );
```

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***.

rnut_gsize

Returns a number of siblings of the calling NUT process in its grid.

Synopsis

```
gridsize := rnut_gsize()
```

Return values

`gridsize` Integer numeric object returning a number of siblings of the calling NUT process in its grid.

Description

The routine ***rnut_gzise*** returns a number gridsize of NUT processes, started by the ***rnut_spawn*** routine, which has started the requesting process. The number ***gridsize*** includes the requesting process, but does not include the parent process. An array of sibling tids can be determined by ***rnut_mygrid***(3NUT) routine.

Example

```
size := rnut_gsize();
```

Errors

rnut_gzise returns ***nil*** in error situations. In the open user interaction mode, on failure, the routine displays a message window to notify the user about the error. In the closed user interaction mode the routine prints an error message to ***stderr***.

rnut_kill

Kills remote NUT processes.

Synopsis

```
rnut_kill( tids, nnut )
```

Arguments

tids	Numeric array of length ***nnut*** containing tids of NUT processes to kill. It can contain a tid of the calling NUT.
nnut	Integer numeric object specifying a number of NUT processes to kill.

Description

The routine ***rnut_kill*** is used to kill ***nnut*** remote NUT processes identified by ***tids***.

When all of running NUT processes have exited or have been killed the NUT daemon ***nutd*** deletes all hosts which were added to a configuration of PVM from a list of hosts allowed to connect to the root X server. Then ***nutd*** halts PVM.

Example

```
tids:= rnut_spawn(3, 2);
..
rnut_kill( tids, length(tids));
```

Errors

In the open user interaction mode, on failure (e.g. bad parameters), ***rnut_kill*** displays a window to notify the user about the error. In the closed user interaction mode ***rnut_kill*** prints an error message to ***stderr***.

rnut_mygrid

Returns the tids of the sibling NUT processes.

Synopsis

```
stids := rnut_mygrid()
```

Return values

stids Numeric array formed of the tids of the NUT/PVM processes started by the same rnut_spawn routine, as the calling process.

Description

The routine ***rnut_mygrid*** returns an array ***stids*** of the PVM(3PVM) task identifiers for each NUT started by the same ***rnut_spawn*** routine, as the calling process. The array ***stids*** includes tid of the calling process, but does not include tid of the parent process. A number of sibling can be determined by ***length*** or ***rnut_gsize***(3NUT) routines. The PVM task identifier of the parent process can be determined by ***rnut_parent***(3NUT) routine.

Example

```
stids := rnut_mygrid();
rnut_xstore(stids, length(stids), 'xx', x);
a := rnut_ifetch(rnut_parent(), 'input');
```

Errors

rnut_mygrid returns ***nil*** in error situations. In the open user interaction mode, on failure, the routine displays a message window to notify the user about the error. In the closed user interaction mode the routine prints an error message to ***stderr***.

rnut_myhost

Returns the host of the calling NUT process.

Synopsis

```
myhost := rnut_myhost()
```

Return values

myhost Text object returning the name of the host on which the calling NUT process is running.

Description

The routine ***rnut_myhost*** returns the name of the host on which the calling NUT process is located.

Example

```
%spawn my children on my host
myhost := rnut_myhost();
tids := rnut_spawn(3, 2, myhost);
```

rnut_mymode

Returns a value of the user interaction mode of the calling NUT process.

Synopsis

```
mymode := rnut_mymode()
```

Return values

mymode Numeric object returning the value of the user interaction mode of the calling NUT processes. ***mode*** can have one of the values:

Mode value	MEANING
0	NUT is running with open window interface (i.e. root NUT process)
1	NUT is running outside the PVM
2	NUT is running with open window interface
3	NUT is running with closed window interface

Description

The routine ***rnut_mymode*** returns the value of the user interaction mode of the calling NUT process. This value can be used for changing the mode of the calling NUT process.

Example

```
if rnut_mymode() == 2 ->
     rnut_chmod([rnut_mytid()], 1, 3); % close my window
fi;
```

Errors

rnut_mymode returns ***nil*** in error situations. In the open user interaction mode, on failure, the routine displays a message window to notify the user about the error. In the closed user interaction mode the routine prints an error message to ***stderr***.

rnut_mynum

Returns the instance number of the calling NUT process in the group of its siblings, i.e. in its grid.

Synopsis

```
mynum := rnut_mynum()
```

Return values

`mynum` Numeric object returning instance number of the calling process in the group of its siblings, i.e. in its grid.

Description

The routine ***rnut_mynum*** returns a unique instance number of the calling NUT processin its grid, i.e. in the group of NUT processes which were spawned by the same routine ***rnut_spawn***(3NUT). Instance numbers start at 1 and count up. Zero number can be used as an instance number of a parent NUT which spawned the grid.

The routine ***rnut_mynum*** can be called multiple times in NUT application. On its first call from the root NUT process, ***rnut_mynum*** starts PVM daemon ***pvmd3***(3PVM), if it has not been started already, then starts NUT daemon ***nutd*** on the host, displaing the NUT window interface, as well as adds all PVM hosts to the list of hosts allowed to connect to the X server.

Called from the root NUT ***rnut_mynum*** returns 1. If the calling NUT process was created by ***rnut_spawn***(3NUT), then ***rnut_mynum*** simply returns its instance number in the group of spawned processes.

Example

```
stids:= rnut_mygrid();
mynum := rnut_mynum();
mya:= rnut_xfetch(stids[mynum + 1], 'a');
```

Errors

rnut_mynum returns ***nil*** in error situations. In the open user interaction mode, on failure, the routine displays a message window to notify the user about the error. In the closed user interaction mode the routine prints an error message to ***stderr***.

rnut_mypack

Returns a name of the package loaded to the calling NUT process.

Synopsis

```
mypack := rnut_mypack()
```

Return values

mypack Text object returning the name of the package loaded to the calling NUT process.

Description

The routine ***rnut_mypack*** returns the name of the package loaded to the calling NUT process. This name can be used to test, send and change a package name.

Examples

```
me := rnut_mytid();
if rnut_mypack() == ‘eda.mem’ ->
     rnut_pack([me], 1, ‘next.mem’); % change my package
fi;
..
tids := rnut_spawn(3, 2);
%send the name of my package to my children
rnut_pack(tids, length(tids), rnut_mypack())
```

rnut_mytid

Returns the tid of the calling NUT process.

Synopsis

```
mytid := rnut_mytid()
```

Return values

`mytid` Numeric object returning the task identifier of the calling NUT process.

Description

The routine ***rnut_mytid*** returns a unique PVM tid of the calling NUT process. ***rnut_mytid*** can be called multiple times in NUT application. On its first call from the root NUT process, ***rnut_mytid*** starts PVM daemon ***pvmd3***(3PVM), if it has not been started already, then starts NUT daemon ***nutd*** on the host, displaing the NUT window interface, as well as adds all PVM hosts to the list of hosts allowed to connect to the X server. If the calling NUT process was created by ***rnut_spawn***(3NUT), then ***rnut_mytid*** simply returns its tid.

Examples

```
mytid := rnut_mytid();
. .
x := rnut_xfetch(rnut_mytid(), 'xx');
```

Errors

rnut_mytid returns ***nil*** in error situations. In the open user interaction mode, on failure, the routine displays a message window to notify the user about the error. In the closed user interaction mode the routine prints an error message to ***stderr***.

rnut_pack

Sends a name of a package to remote NUT processes and asks to open it.

Synopsis

```
info := rnut_pack( tids, nnut, package )
```

Arguments

`tids`	Numeric array of length ***nnut*** containing tids of destination NUT processes. It can contain a tid of the calling NUT.
`nnut`	Integer numeric object specifying a number of NUT processes to send the package name.
`package`	Text object specifying the name of the package to open in the destination NUT processes.

Return values

`info`	The routine returns the name ***package*** in ***info***. On failure, it returns ***nil***.

Description

The routine ***rnut_pack*** is used to open a package named ***package*** in remote NUT processes. It sends the name of the package **package** to ***nnut*** NUT processes identified by ***tids*** with a request to load this package for processing. Each receiving NUT opens this package. If remote NUT already works with another loaded package it displays a window asking a permission to change the package.

Example

```
tids:= rnut_spawn(3, 2);
rnut_clcpy( tids, length(tids), 'grid', 'ch_grid');
```

Errors

rnut_pack returns ***nil*** in error situations. In the open user interaction mode, on failure, ***rnut_pack*** displays a window to notify the user about the error. In the closed user interaction mode ***rnut_pack*** prints an error message to ***stderr***.

rnut_parent

Returns the tid of NUT that spawned the calling process.

Synopsis

```
parent:= rnut_parent()
```

Return values

parent Numeric object returning the task identifier of the parent NUT of the calling process. If the calling NUT is root, i.e. it was not created with ***rnut_spawn***(3NUT), then ***rnut_parent*** returns value -23. This value means error condition PvmNoParent (see ***pvm_parent***(3PVM)).

Description

The routine ***rnut_parent*** returns a PVM tid of NUT that spawned the calling process. ***rnut_parent*** can be called multiple times in NUT application. On its first call from the root NUT process, ***rnut_parent*** starts PVM daemon ***pvmd3***(3PVM), if it has not been started already, then starts NUT daemon ***nutd*** on the host, displaing the NUT window interface, as well as adds all PVM hosts to the list of hosts allowed to connect to the X server. Calling from the root NUT rnut_parent returns the value -23, that means error condition PvmNoParent. If the calling NUT process was created by ***rnut_spawn***(3NUT), then ***rnut_parent*** simply returns a tid of its parent.

Examples

```
parent:= rnut_parent();
. .
rnut_xstore([rnut_parent()], 1, 'xx', x);
```

Errors

The routine returns ***nil*** in error situations or -23 if the calling protcess has not a parent, i.e. it was not spawned with rnut_spawn(3NUT) routine. In the open user interaction mode, on failure, the routine displays a message window to notify the user about the error. In the closed user interaction mode the routine prints an error message to ***stderr***.

rnut_perform

Multicasts a perform request to remote NUT processes.

Synopsis

```
rnut_perform( tids, nnut )
```

Arguments

tids Numeric array of length ***nnut*** containing tids of destination NUT processes. It can contain a tid of the calling NUT.

nnut Integer numeric object specifying a number of NUT processes to send the perform request.

Description

The routine ***rnut_perform*** multicasts the perform request to ***nnut*** NUT processes identified by ***tids***. Each receiving NUT executes current NUT code in its Workspace.

Example

```
% spawn 3 children with closed windows
% copy the class 'grid' to children
% multicast the workspace to children
% perfrom the workspace in children
tids:= rnut_spawn(3, 3);
rnut_clcpy(tids, 3, 'grid', 'grid');
rnut_putwsr( tids, length(tids), '
         % this is workspace for children:
         tids := rnut_spawn(2, 3);
         rnut_clcpy(tids, 2, ''grid'', ''grid'');
');
rnut_perform(tids, 3);
```

Errors

In the open user interaction mode, on failure (e.g. bad parameters), ***rnut_perform*** displays a window to notify the user about the error. In the closed user interaction mode ***rnut_perform*** prints an error message to ***stderr***.

rnut_pgetwsa, rnut_pgetwsr

Get a copy of the text of the Workspace from remote NUT process and perform it.

Synopsis

```
rnut_pgetwsa( tid )
rnut_pgetwsr( tid )
```

Arguments

tid Integer numeric objec specifying a tid of the remote NUT process.

Description

Routine ***rnut_pgetwsa*** and ***rnut_pgetwsr*** are used to get a copy of the text of the Workspace from the remote NUT specified by ***tid***, to place it into the Workspace of the calling NUT and then perform it. The routines send a request to get back a text of the Workspace from the remote NUT.

Routines are asynchronous, i.e. computation in the calling NUT continues as soon as the request to get the Workspace is sent to the remote NUT. Whenever the text of the Workspace is arrived the calling NUT appends it to its current Workspace in the case of ***rnut_getwsa*** and then performs it. In the case of ***rnut_pgetwsr*** the calling NUT process replaces its current Workspace by the new Workspace from remote NUT and then performs it.

Example

```
rnut_pgetwsa( rnut_parent() );
```

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***.

rnut_pputwsa, rnut_pputwsr

Multicast the text for the Workspace to remote NUT processes with the request to perform it.

Synopsis

```
rnut_pputwsa( tids, nnut, ws )
rnut_pputwsr( tids, nnut, ws )
```

Arguments

`tids` Numeric array of length ***nnut*** containing tids of distination NUT processes. It can contain a tid of the calling NUT.

`nnut` Integer numeric object specifying a number of NUT processes to send the text for Workspace and the perform request.

`ws` Text object specifying the text for the Workspace to mulicast to the destination NUT processes.

Description

Routines ***rnut_pputwsa*** and ***rnut_pputwsr*** multicast the text for Workspace to ***nnut*** NUT processes identified by ***tids*** with the request to perform it.

rnut_pputwsa sends the text ***ws*** to specified remote NUT processes with request to append this text to the current text in their Worspace and perform text ws. Each receiving NUT appends text ***ws*** to its Workspaceand then performs text ***ws***.

rnut_putwsr sends the text ***ws*** to specified remote NUT processes with request to replace the current text in their Worspace by the text ***ws*** and perform it. Each receiving NUT replaces its Workspace by the new text ***ws*** and then performs it.

Example

```
tids:= rnut_spawn(3, 3, nil, 'grid.mem');
rnut_pputwsr( tids, 3, '
          g:= new mygrid;
          NutPrint(''I was born just now!'');
');
rnut_pputwsa( [tids[2]], 1, '
          g.clget(rnut_parent(), 'adder', 'adder');
');
```

Errors

In the open user interaction mode, on failure (e.g. bad parameters), routines display a window to notify the user about the error. In the closed user interaction mode rnut_perform print an error message to ***stderr***.

rnut_putwsa, rnut_putwsr

Multicast the text for the Workspace to remote NUT processes.

Synopsis

```
rnut_putwsa( tids, nnut, ws )
rnut_putwsr( tids, nnut, ws )
```

Arguments

`tids`	Numeric array of length ***nnut*** containing tids of distination NUT processes. It can contain a tid of the calling NUT.
`nnut`	Integer numeric object specifying a number of NUT processes to send the text for Workspace.
`ws`	Text object specifying the text for the Workspace to mulicast to the destination NUT processes.

Description

Routines ***rnut_putwsa*** and ***rnut_putwsr*** multicast the text for Workspace to ***nnut*** NUT processes identified by ***tids***.

rnut_putwsa sends the text ***ws*** to specified remote NUT processes with request to append this text to the current text in their Worspace. Each receiving NUT appends text ***ws*** to its Workspace.

rnut_putwsr sends the text ***ws*** to specified remote NUT processes with request to replace the current text in their Worspace by the text ***ws***. Each receiving NUT replaces its Workspace by the new text ***ws***.

Example

```
tids:= rnut_spawn(3, 3, nil, 'grid.mem');
rnut_putwsr( tids, 3, '
        g:= new mygrid;
        NutPrint(''I was born just now!'');
');
rnut_putwsa( [tids[2]], 1, '
        g.clget(rnut_parent(), 'adder', 'adder');
');
```

Errors

In the open user interaction mode, on failure (e.g. bad parameters), routines display a window to notify the user about the error. In the closed user interaction mode rnut_perform print an error message to ***stderr***.

rnut_spawn

Starts new NUT processes on PVM(3PVM).

Synopsis

```
tids := rnut_spawn( nnut, mode, where, package )
```

Arguments

`nnut` Integer numeric object specifying the number of copies of NUT to start.

`mode` Integer numeric object specifying the initial user interaction mode of NUT processes. ***mode*** should have one of the values:

Mode value	MEANING
nil	Start NUT in default mode **2**
1	Start NUT outside the PVM
2	Start NUT with open window interface (default)
3	Start NUT with closed window interface

`where` Text object specifying a host name where to start new NUT processes. ***where*** can be ***nil*** or a host name such as 'sole.electrum.kth.se'. If ***where*** is ***nil***, then PVM will select the most appropriate host like in ***pvm_spawn***(3PVM) routine.

`package` Text object specifying a package name for the spawned NUT processes. ***package*** can be ***nil*** or a package name, e.g. 'eda.mem'. If ***package*** is ***nil***, then each of spawned NUT processes starts with an untitled empty package. If ***package*** does not exist, then each of spawned NUT processes also starts with empty package, but names it ***package***.

Return values

`tids` Numeric array of length ***nnut*** formed of the tids of the NUT/PVM processes started by this ***rnut_spawn*** routine.

Description

The routine ***rnut_spawn*** starts ***nnut*** copies of NUT.

The hosts on which the NUT processes are started are determined by the ***where*** argument. On success, ***rnut_spawn*** returns a numeric array ***tids*** of the PVM task identifiers for each NUT started. Otherwise, routine returns ***nil***.

If the host specified by ***where*** argument is not in PVM the routine adds it to the configuration of PVM and to the list of hosts allowed to connect to the root X server. It is, however, more convinient to use ***rnut_spawn*** with ***where*** being ***nil***, provided a configuration of PVM has been defined before by means of ***rnut_addhosts*** and ***rnut_delhosts***(3NUT) routines.

rnut_spawn uses ***pvm_catchout***(3PVM) to catch output from spawned NUT processes, to tag each line by tid of the child NUT and to print it to the ***stderr*** or ***stdout*** files of the root NUT.

Examples

```
% start two children with open windows on 'bream...'
% with the package 'p.mem'
    tids := rnut_spawn(2, 2, 'bream.electrum.kth.se', 'p.mem');
% add two hosts into PVM
% start 4 children with close windows somewhere
    rnut_addhosts(2, ['whale', 'cod']);
    tids := rnut_spawn(4, 3, nil, nil);
% start child with the package 'mypackage'
    tids := rnut_spawn(1, 2, nil, 'mypackage');
```

Errors

rnut_spawn returns nil in error situations. In the open user interaction mode, on failure, ***rnut_spawn*** displays a message window to notify the user about the error. In the closed user interaction mode ***rnut_spawn*** prints an error message to ***stderr***.

rnut_*store

Store an object to a set of NUT processes.

Synopsis

```
info := rnut_xstore( tids, nnut, objname, obj )
info := rnut_sstore( tids, nnut, objname, obj )
info := rnut_istore( tids, nnut, objname, obj )
info := rnut_ustore( tids, nnut, objname, obj )
```

Arguments

tids Numeric array of length ***nnut*** containing the tids of NUT processes to be stored to. It can contain the tid of the sending NUT process.

nnut Integer numeric object specifying the number of NUT processes which receive values.

objname Text object containing the name of the object to be stored into.

obj Object to be stored.

Return values

info Each of the store routines returns the object ***obj*** in ***info***. On failure, they return ***nil***.

Description

These are storing routines of a general-purpose communication and synchronization package, based on the parallel model, called ***EDA***, designed at the Department of Teleinformatics of the Royal Institute of Technology, Sweden []. They can be used to store a value of the object ***obj*** into objects, named ***objname***, of the NUT processes, specified by array ***tids***. These NUT processes can be spawned by ***rnut_spawn***(3NUT).

rnut_xstore is a blocking communication routine supporting mutually exclusive interprocess communication. It sends a value of an object ***obj*** to ***nnut*** NUT processes specified by ***tids*** with a request to store it into their objects named ***objname***. The routine blocks the sending NUT process until ***nnut*** acknowledgments have arrived from all tids. Each of the receiving NUT processes sends an acknowledgment as soon as the store operation has been performed. Each receiving process can execute store request only if its object named ***objname*** is empty, i.e. has ***nil*** value or does not exist. In the last case receiving process creates an object named ***objname*** from the same class as ***obj***. Moreover, if receiving process does not have that class, then it gets the class from sender and compiles it. If the object named ***objname*** is not empty, then the store request is queued until the ***objname*** becomes emptied by the matching ***rnut_xfetch***(3NUT) or ***rnut_sfetch***(3NUT). Once ***nnut*** acknowledgments have arrived, computation on the sending process resumes.

rnut_sstore is a non-blocking communication routine supporting object streams between NUT processes. It sends a value of an object ***obj*** to ***nnut*** NUT processes specified by ***tids*** with a request to store it into their objects named ***objname***. Computation on the sending process continues. Each receiving NUT process can execute store request only if its object named ***objname*** is empty, i.e. has ***nil*** value or does not exist. Otherwise the store request is queued until the ***objname*** becomes emptied the matching ***rnut_xfetch***(3NUT) or ***rnut_sfetch***(3NUT).

rnut_istore is a non- blocking routine supporting write-once shared objects. It sends a value of an object ***obj*** to nnut NUT processes specified by ***tids*** with a request to store it into their objects named ***objname***. Computation on the sending process continues. Each receiving NUT process can execute store request only if its object named ***objname*** does is empty, i.e. has ***nil*** value or does not exist. Otherwise the store request is ignored.

rnut_ustore is non-blocking communication routine supporting unconditional updating of shared objects. It sends a value of an object ***obj*** to nnut NUT processes specified by ***tids*** with a request to store it into their objects named ***objname***. Computation on the sending process continues. Each receiving NUT process always executes store request independently on status of the object named ***objname***.

Examples

```
tids := rnut_mygrid();
n := length(tids);
info := rnut_xstore(tids, n, 'pi', 3.1415926);
. .
p := [rnut_parent()];
rnut_sstore(p, 1, 'stream', 'First string');
rnut_sstore(p, 1, 'stream', 'Second string');
. .
b := new myclass;
b.input := [-1.3, 'minus one point three'];
rnut_ustore([rnut_mytid()], 1, 'ss', b);
```

Errors

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

Conclusion

The approach to parallel programming taken in this project combines distributed computing in a heterogeneous network of computers with automatic synthesis of programs. Using the extended dataflow actor model as a program model for distributed computing seems to be very promising. This can be used for programming of parallel computations on various platforms and with various levels of abstraction. Finally, this can lead to a unified language for specification of hardware and software for parallel computing.

The functions developed in this report support coarse-grained parallel computing, and combined with the program synthesis in NUT, can be used for programming in a compositional way. An example of a parallel search program presented in the Appendix 2 shows the feasibility of this approach. After developing a collection of classes for describing search problems, a particular problem could be specified on a half of a page, using these classes. Sets of high-level control structures in the form of NUT classes must be developed for various application areas in order to make the present toolkit attractive for users. This, together with development of visual programming tools for distributed computing, can be the next step of the research in parallel programming at the CSlab of the Teleinformatics Department of the KTH.

Acknowledgments

We wish to express our thanks to Mari Kopp for her patience in consulting us on the matters of internal structures of NUT. We are thankful to Urmas Kopra who was the first to implement parallel processes in the NUT environment and to test out the usage of NUT on top of PVM. This work could not been done in the present form without the ideas developed in the parallel architectures group of the Teleinformatics Department and we wish to thank especially Lars-Erik Thorelli and Hallo Ahmed for the discussions on the EDA model. We have a pleasure to thank Tarmo Uustalu for many useful discussions on the design of rNUT Toolkit.

References

1. E Tyugu. “*The NUT system*”.Teleinformatics/KTH 1994 (Internal report).

2. T. Uustalu, U. Kopra, V. Kotkas, M. Matskin and E Tyugu. “*The NUT Language Report*”. Technical Report TRITA-IT-R 94:14, Dept. of Teleinformatic, KTH. 1994.

3. L-E Thorelli, *“The EDA Multiprocessing Model”*. Technical Report TRITA-IT-R 94:28, CSLab, Dept. of Teleinformatic, KTH. 1994.

4. V.V. Vlassov., L-E Thorelli and H. Ahmed, *“Multi- EDA A Programming Environment for Parallel Computations”*. Technical Report TRITA-IT-R 94:29, CSLab, Dept. of Teleinformatic, KTH. 1994.

5. G.A. Geist, et al, *“PVM3 User’s Guide and Reference Manual”*. ORNL/TM-12187, Oak Ridge National Lab. May 1994.

6. V.S. Sunderam, G.A. Geist, J. Dongarra and R. Manchek, *“The PVM Concurrent Computing System: Evaluation, Experiences, and trends”*. Parallel Computing, Vol. 20, No. 4, April 1994, pp. 531-546.

7. U. Kopra. *“Parallel Implementation of Synthesized Programs in NUT“*. Teleinformatics/KTH 1994 (Internal report).

Appendix A1: Grid and Mygrid Classes

This Appendix contains examples of texts of *grid* and *mygrid* classes. Objects from these classes can be used to simplify an addressing scheme to members of a grid which are spawned with *rnut_spawn* routine. Each member of the grid, i.e. the parent and each child, obtains a number which is used for addressing with relations from both classes. The class *grid* can be used in the parent process, class *mygrid* - in a child. The parent NUT process obtains number 0, children are numbered from 1 to n. The first relation *gettids* in both classes is used to transform an array of numbers (*nodes* in these examples) into an array of tids. All other relations use with rNUT routines node numbers, but not tids.

These classes provide a library of rNUT routines which use node number to specify a destination remote NUT process instead of its tid.

The following examples illustrate a usage of objects of *grid* and *mygrid* classes.

Example 1.

```
% SPAWN
g:=new grid;
tids := g.spawn(3, 2);
t := [1, 2, 3];
% CLOSE / OPEN
g.close(t);
g.open([1, 2]);
% WORK SPACE PUT
g.putwsr([2],`
        myg:=new mygrid;
        g:=new grid;
        g.spawn(2, 2);
        t  := [1,2];
        g.pack(t,''child1.mem'');`);
g.perform([2]);
g.pputwsr([2], `g.exit(t);');
% EDA XSTORE
b:=new grid 1, 2, `sole.electrum.kth.se';
g.xstore([1], `aa', b);
g.xstore([1], `aa', `Test string');
g.xstore([1], `aa', [`first', `second', `third']);
%EXIT
tt:=g.tids;
rnut_exit(tt, length(tt));
```

Example 2.

```
myg := new mygrid;
myg.clget(0, `grid`);
me := myg.mynum;
g.ustore([me - 1, me + 1], 2, `pi', 3.1415926);
newg := new grid;
newg.spawn(4);
g.sstore([3], `xx', myg);
g.sstore([3], `xx', 5);
g.sstore([3], `xx', -1.3);
g.sstore([3], `xx', `Hello world');
g.sstore([3], `xx', [7, 8]);
g.sstore([3], `xx', [`radio-', `ga-', `ga-']);
g.pputwsr([3] ,'
        x := new array of any;
        for i := 1 to 6 do
                x[i] := rnut_xfetch(rnut_mytid(), ''xx'');
        od;
        rnut_chmod([rnut_mytid()], 1, 2);
`);
```

A1.1 Class grid

```
var
        size:   num;            % size of remote NUT grid
        imode:  num;            % initial mode of remote NUTs
                                %       0 - Rnut_X (root Nut, here not used! )
                                %       1 - Rnut_onlyX (only X_main_loop)
                                %       2 - Rnut_openX (PVM & X)
                                %       3 - Rnut_closeX (PVM & unvisible_X)
        ihost:  text;           % initial host to spawn remote NUTs
        ipack:  text;           % initial pack to spawn remote NUTs

        tids:   array of num;% tids of child rNUT in the grid

        mytid:  num;            % my tid
        myhost: text;           % my host
        mymode: num;            % my mode
        mypack: text;           % my pack

        node:   num;            % number of node in subgrid to operate
        nodes:  array of num;% numbers of nodes in subgrid to operate:
                                %       0 - me
                                %       1 ... size - rNUT
        name:   text;           % input object name
        input:  any;            % input object
        output: any;            % output object
        wtids:  array of num;% array of tids to operate
init
        mytid   := rnut_mytid();
        mymode  := rnut_mymode();
        myhost  := rnut_myhost();
        mypack  := rnut_mypack();
        node    := 0;
rel
        gettids: nodes, mytid, tids -> wtids {
                wtids := new array of num;
                i := 1;
                do (i <= length(nodes)) & (nodes[i] /= nil) ->
                        if nodes[i] <= 0 ->
                                wtids[i] := mytid;
                                i := i + 1;
                        ||  true ->
                                n <- nodes[i];
                                wtids[i] := tids[n];
                                i := i + 1;
                        fi;
                od;
        };
        spawn:  size, imode, ihost, ipack -> tids  {
                tids := rnut_spawn( size, imode, ihost, ipack );
                };
        exit: nodes, mytid, tids, wtids -> output, wtids {
                wtids := gettids(nodes, mytid, tids);
                output := rnut_exit(wtids, length(wtids));
        };
        open: nodes, mytid, tids, wtids -> output, wtids {
                wtids := gettids(nodes, mytid, tids);
                output := rnut_chmod(wtids, length(wtids), 2);
        };
        close: nodes, mytid, tids, wtids -> output, wtids {
                wtids := gettids(nodes, mytid, tids);
                output := rnut_chmod(wtids, length(wtids), 3);
        };
        gethostname: nodes, mytid, tids, wtids -> output, wtids {
                wtids := gettids(nodes, mytid, tids);
```

```
        output := rnut_gethost(wtids, length(wtids));
};
pack: nodes, input, mytid, tids, wtids -> output, wtids {
        wtids := gettids(nodes, mytid, tids);
        output := rnut_pack(wtids, length(wtids), input);
};
sendclass: nodes, input, mytid, tids, wtids -> output, wtids {
        wtids := gettids(nodes, mytid, tids);
        output := rnut_clcpy(wtids, length(wtids), input, input);
};
getclass: node, input, mytid, tids -> output {
        if node <= 0 -> output := rnut_clget(mytid, input, input);
        || true ->      output := rnut_clget(tids[node], input, input)
        fi
};
putwsa: nodes, input, mytid, tids, wtids -> output, wtids {
        wtids := gettids(nodes, mytid, tids);
        output := rnut_putwsa(wtids, length(wtids), input);
};
putwsr: nodes, input, mytid, tids, wtids -> output, wtids {
        wtids := gettids(nodes, mytid, tids);
        output := rnut_putwsr(wtids, length(wtids), input);
};
pputwsa: nodes, input, mytid, tids, wtids -> output, wtids {
        wtids := gettids(nodes, mytid, tids);
        output := rnut_pputwsa(wtids, length(wtids), input);
};
pputwsr: nodes, input, mytid, tids, wtids -> output, wtids {
        wtids := gettids(nodes, mytid, tids);
        output := rnut_pputwsr(wtids, length(wtids), input);
};
getwsa: node, input, mytid, tids -> output {
        if node <= 0 -> output := rnut_getwsa(mytid);
        || true ->output := rnut_getwsa(tids[node])
        fi
};
getwsr: node, input, mytid, tids -> output {
        if node <= 0 -> output := rnut_getwsr(mytid);
        || true ->output := rnut_getwsr(tids[node])
        fi
};
pgetwsa: node, input, mytid, tids -> output {
        if node <= 0 -> output := rnut_pgetwsa(mytid);
        || true ->output := rnut_pgetwsa(tids[node])
        fi
};
pgetwsr: node, input, mytid, tids -> output {
        if node <= 0 -> output := rnut_pgetwsr(mytid);
        || true ->output := rnut_pgetwsr(tids[node])
        fi
};
perform: nodes, mytid, tids, wtids -> output, wtids {
        wtids := gettids(nodes, mytid, tids);
        output := rnut_perform(wtids, length(wtids));
};
xstore: nodes, name, input, mytid, tids, wtids -> output, wtids {
        wtids := gettids(nodes, mytid, tids);
        output := rnut_xstore(wtids, length(wtids), name, input);
};
sstore: nodes, name, input, mytid, tids, wtids -> output, wtids {
        wtids := gettids(nodes, mytid, tids);
        output := rnut_sstore(wtids, length(wtids), name, input);
};
istore: nodes, name, input, mytid, tids, wtids -> output, wtids {
        wtids := gettids(nodes, mytid, tids);
```

```
        output := rnut_istore(wtids, length(wtids), name, input);
};
ustore: nodes, name, input, mytid, tids, wtids -> output, wtids {
        wtids := gettids(nodes, mytid, tids);
        output := rnut_ustore(wtids, length(wtids), name, input);
};
xfetch: node, name, mytid, tids -> output {
        if node <= 0 -> output := rnut_xfetch(mytid, name);
        || true ->output := rnut_xfetch(tids[node], name)
        fi
};
ifetch: node, name, mytid, tids -> output {
        if node <= 0 -> output := rnut_ifetch(mytid, name);
        || true ->output := rnut_ifetch(tids[node], name)
        fi
};
sfetch: node, name, mytid, tids -> output {
        if node <= 0 -> output := rnut_sfetch(mytid, name);
        || true ->output := rnut_sfetch(tids[node], name)
        fi
};
ufetch: node, name, mytid, tids -> output {
        if node <= 0 -> output := rnut_ufetch(mytid, name);
        || true ->output := rnut_ufetch(tids[node], name)
        fi
};
```

A1.2 Class mygrid

```
var
        size:   num;            % size of remote NUT grid
        parent: num;            % tid of parent
        tids:   array of num;   % tids of rNUTs in the grid, including me,
                                % but excluding parent

        mytid:  num;            % my tid
        myhost: text;           % my host
        mymode: num;            % my mode
        mypack: text;           % my pack
        mynode: num;    % my number

        node:   num;            % number of rNUT to operate:
                                %       0 - parent
                                %       1 ... size - rNUTs including me
        nodes:  array of num;   % array of nodes to operate
        name:   text;           % input object name
        input:  any;            % input object
        output: any;            % output object
        wtids:  array of num;% array of tids to operate
init
        size := rnut_grsize() + 1;
        parent := rnut_parent();
        tids := rnut_mygrid();
        mynode := rnut_mynum();
        mypack := rnut_mypack();
        mymode := rnut_mymode();
        mytid := rnut_mytid();
        myhost := rnut_myhost();
rel
        gettids: nodes, parent, tids -> wtids {
                for i:= 1 to length(nodes) do
                        if nodes[i] <= 0 ->
                                wtids[i] := parent;
                        || true ->
```

```
                        n <- nodes[i];
                        wtids[i] := tids[n]
                fi
        od;
};
open: nodes, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_chmod(wtids, length(wtids), 2);
};
close: nodes, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_chmod(wtids, length(wtids), 3);
};
gethostname: nodes, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_gethost(wtids, length(wtids));
};
pack: nodes, input, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_pack(wtids, length(wtids), input);
};
sendclass: nodes, input, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_clcpy(wtids, length(wtids), input, input);
};
getclass: node, input, parent, tids -> output {
        if node <= 0 -> output := rnut_clget(parent, input, input);
        || true ->      output := rnut_clget(tids[node], input, input)
        fi
};
putwsa: nodes, input, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_putwsa(wtids, length(wtids), input);
};
putwsr: nodes, input, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_putwsr(wtids, length(wtids), input);
};
pputwsa: nodes, input, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_pputwsa(wtids, length(wtids), input);
};
pputwsr: nodes, input, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_pputwsr(wtids, length(wtids), input);
};
getwsa: node, input, parent, tids -> output {
        if node <= 0 -> output := rnut_getwsa(parent);
        || true ->output := rnut_getwsa(tids[node])
        fi
};
getwsr: node, input, parent, tids -> output {
        if node <= 0 -> output := rnut_getwsr(parent);
        || true ->output := rnut_getwsr(tids[node])
        fi
};
pgetwsa: node, input, parent, tids -> output {
        if node <= 0 -> output := rnut_pgetwsa(parent);
        || true ->output := rnut_pgetwsa(tids[node])
        fi
};
pgetwsr: node, input, parent, tids -> output {
        if node <= 0 -> output := rnut_pgetwsr(parent);
        || true ->output := rnut_pgetwsr(tids[node])
        fi
```

```
};
perform: nodes, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_perform(wtids, length(wtids));
};
xstore: nodes, name, input, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_xstore(wtids, length(wtids), name, input);
};
sstore: nodes, name, input, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_sstore(wtids, length(wtids), name, input);
};
istore: nodes, name, input, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_istore(wtids, length(wtids), name, input);
};
ustore: nodes, name, input, parent, tids, wtids -> output, wtids {
        wtids := gettids(nodes, parent, tids);
        output := rnut_ustore(wtids, length(wtids), name, input);
};
xfetch: node, name, parent, tids -> output {
        if node <= 0 -> output := rnut_xfetch(parent, name);
        || true -> output := rnut_xfetch(tids[node], name)
        fi
};
ifetch: node, name, parent, tids -> output {
        if node <= 0 -> output := rnut_ifetch(parent, name);
        || true -> output := rnut_ifetch(tids[node], name)
        fi
};
sfetch: node, name, parent, tids -> output {
        if node <= 0 -> output := rnut_sfetch(parent, name);
        || true -> output := rnut_sfetch(tids[node], name)
        fi
};
ufetch: node, name, parent, tids -> output {
        if node <= 0 -> output := rnut_ufetch(parent, name);
        || true -> output := rnut_ufetch(tids[node], name)
        fi
};
```

Appendix A2: A Parallel Search Problem

In this appendix, we present an example of distributed computing for solving a search problem. The problem is solved by means of breadth-first search. For solving it, a collection of classes is developed. A description of the particular problem taken as an example takes only 15 lines and it can be found at the end of the appendix.

A2.1 Breadth-First Search

An idea of the breadth-first search, BFS, is illustrated in Figure A2.1. The algorithm implemented here has a root process and a process pool containing NUT processes for exploring open nodes of the search tree.

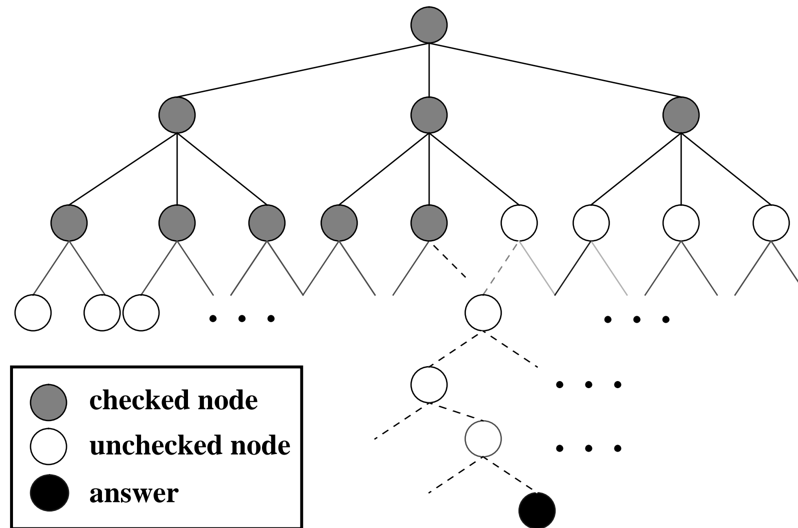

Figure A2.1 Breadth-First Search

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 A2.2).

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 A2.2). Root process gets a job from *queue1* finds a free process in the pool and sends the job to that process (see Figure A2.3). 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 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 notes that a process is free.
3. *answer*: the answer is found and has the shortest path. Root sends exit message to all processes and notify the answer.

Root takes the following actions:

1. Making and administrating the process pool.
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.

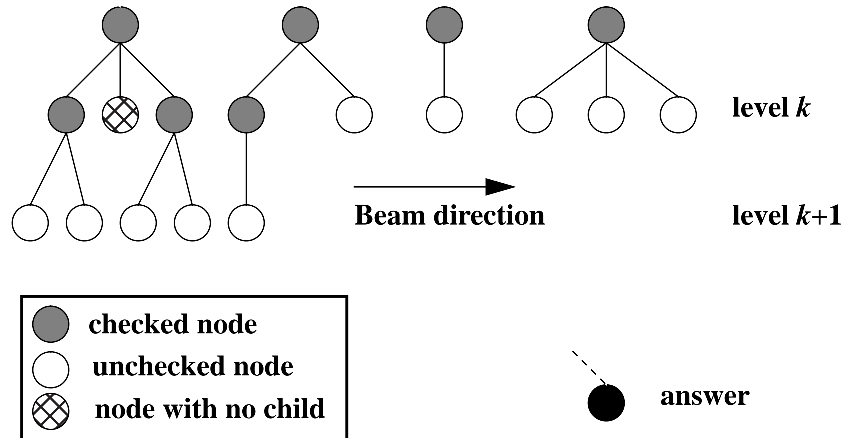

Figure A2.2 Beam Search
(the first beam is searching in level k,
the second beam produces nodes in the next level k+1)

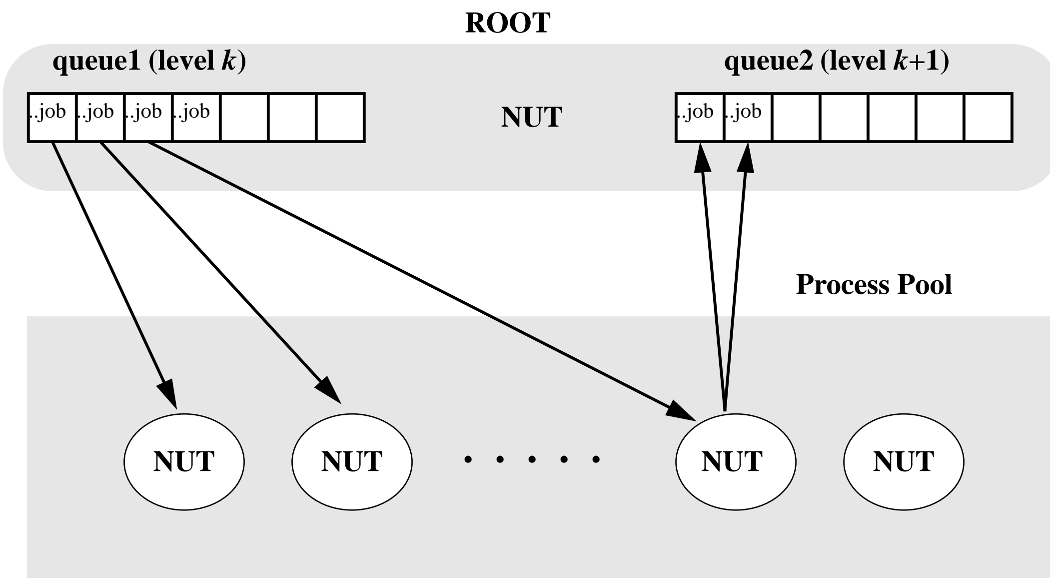

Figure A2.3 A Process Pool with Two Queues

A2.2 Implementation

The package is implemented in NUT. Using rnut functions gives us facilities to use PVM as a message interface and manager between NUT processes which run in a distributed way. In this section we discuss how we used these tools to implement our process pool.

A2.2.1 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 the process pool are managed in other classes and the user don't need to care about them. But she can still choose (through scheme editor) which hosts should be connected to PVM and she can also change the number of processes in process pool.

The program starts in the *main* NUT process, here we choose that this `main` NUT should work as a *monitor* and spawn the search root (see Figure A2.4). This is a good choice because the user can still interact with the package during the root and process pool searching, furthermore if her 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).

```
%################################################################################   %
%                            PARALLEL SEARCH SOFTWARE                              %
%                            A Project in Knowledge-Based Software Technology       %
%                            Royal Institute of Technology, Stockholm               %
%                            Fall 1994                                              %
%################################################################################   %
% Start PVM by selecting 'Compute ALL'under 'Scheme' after performing the following line:
      ClSchemeOpen('PVM');
% Create WorkSpaces for Root & Pool Processes(childs)!
      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 A2.4 Text in the workspace of *monitor*

A2.2.2 Design of Classes

In this part we present the classes of 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:

- *ROOTSTATE*: gives the root state,
- *USRMODIFY*: modifies the state of current nodes and produce children,
- *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:

- *MAKEROOT*: gets all information about root,
- *APPENDPATH*: completes path in the tree of the actual node,
- *ISGOOD*: calls *USRGOOD* to check if the answer is found,
- *MODIFY*: calls *USRMODIFY* to modify the state of the actual node and produces children, checks also duplicated child,
- *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:

- *MAKEPOOL*: creates the process pool and sends their workspaces,
- *STARTJOB*: sends a job to a process pool,
- *FREEPROCESS*: note that a process is free,
- *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:

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

WorkSpaces:

Root & pool processes haven't the same WorkSpaces as the *monitor* NUT. This class contains texts which replace the WorkSpaces of root & pool processes. User can change the number of pool processes by simply changing the WorkSpace of root. This class contains:

- *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.

A2.2.3 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*,...), have been implemented by using this method.

The only object which is being sent between root and *monitor* NUT, is the final answer. Because the answer is unique, it is sent without blocking the `monitor` by storing in a non-blocking way (see Figure A2.5).

A2.3 Test 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

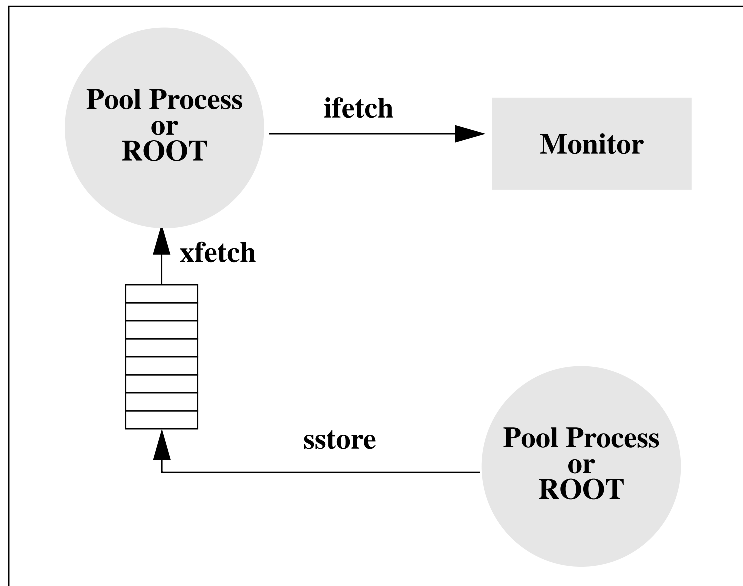

Figure A2.5 Node Communication

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

```
super
     ParSearch;
vir
     outo : 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 -> outo
          { outo[1] := 2*my_state;
          outo[2] := my_state-2;
          outo[3] := my_state+2; };
```

Figure A2.6 shows a typical situation on the screen while working with the package. Observe the *monitor* NUT and scheme editor on the top of the figure, the root which has started in “*bream*” and, finally, see a process pool with three processes running in three different hosts.

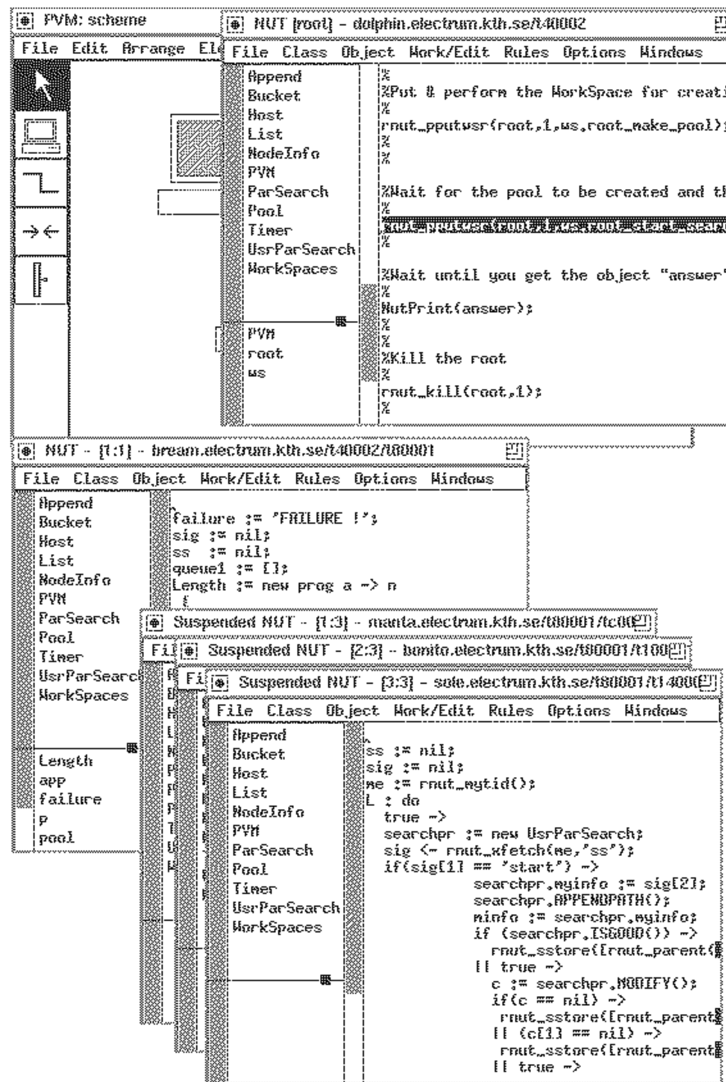

Figure A2.6 NUT Screan

A2.4 Texts of Classes

_WorkSpaces

```
var
        root_make_pool, root_start_search, childs_work_space : text;
init
        root_make_pool :=
        `failure := ``FAILURE !'';
        sig := nil;
        ss := nil;
        queue1 := [];
        Length := new prog a -> n
                { n:=length(a);
                for i to length(a) do
                        if a[i] == nil -> n := i-1; exit; fi;
                od; };
        app := new Append;
        ws := new WorkSpaces;
        pool := new Pool;
        p := pool.MAKEPOOL(3, ws);
        `;
        % ---------------------------------
        root_start_search :=
        `searchpr := new UsrParSearch;
        searchpr.MAKEROOT();
```

```
a := searchpr.mycheck_list;
if (searchpr.ISGOOD()) ->
        rnut_istore([rnut_parent()],1,''answer'',searchpr.mystate);
|| true ->
        c := searchpr.MODIFY();
        if(c == nil) ->
                rnut_istore([rnut_parent()],1,''answer'',failure);
        || true ->
                cinfo := searchpr.CHILDSINFO();
                k := 1;
                do (k <= length(cinfo)) ->
                        if(pool.n_free_process /= 0)->
                                pool.STARTJOB(cinfo[k]);
                        || true ->
                                queue1 := app.APPEND(queue1,cinfo[k]);
                        fi;
                        k := k+1;
                od;
        fi;
fi;
answered_proc := 0;
next_level := 0;
this_level := length(cinfo);
L: do true ->
        if((this_level == 0) & (next_level ==0)) ->
                rnut_istore([rnut_parent()],1,''answer'',failure);
                exit L;
        fi;
        sig <- rnut_xfetch(rnut_mytid(),''ss'');
        if (sig[1] == ``more_childs'') ->
                cinfo := sig[2];
                next_level := next_level + length(cinfo);
                answered_proc := answered_proc + 1;
                for i to length(cinfo) do
                        queue2 := app.APPEND(queue2,cinfo[i]);
                od;
                pool.FREEPROCESS(sig[3]);
        || (sig[1] == ``answer'') ->
                rnut_istore([rnut_parent()],1,''answer'',sig[2]);
                answered_proc := answered_proc + 1;
                answer := sig[2];
                pool.FREEPROCESS(sig[3]);
                exit L;
        || (sig[1] == ``no_more_child'') ->
                answered_proc := answered_proc + 1;
                pool.FREEPROCESS(sig[3]);
        fi;
        if(answered_proc == this_level) ->
                queue1 <- queue2;
                queue2 := [];
                this_level := next_level;
                answered_proc := 0;
                next_level := 0;
        fi;
        do ((pool.n_free_process /= 0) & (queue1[1] /= nil))->
                pool.STARTJOB(queue1[Length(queue1)]);
                queue1[Length(queue1)] := nil;
        od;
od;
for i to (pool.n_process - pool.n_free_process) do
        sig <- rnut_xfetch(rnut_mytid(),''ss'');
od;
pool.EXIT();
`;
% ----------------------------------
childs_work_space :='
ss := nil;
sig := nil;
me := rnut_mytid();
L : do true ->
        searchpr := new UsrParSearch;
        sig <- rnut_xfetch(me,''ss'');
```

```
		if(sig[1] == ''start'') ->
			searchpr.myinfo := sig[2];
			searchpr.APPENDPATH();
			minfo := searchpr.myinfo;
			if (searchpr.ISGOOD()) ->
				rnut_sstore([rnut_parent()],1,''ss'',
					[''answer'',minfo.path_list,sig[3]]);
			|| true -> c := searchpr.MODIFY();
				if(c == nil) ->
					rnut_sstore([rnut_parent()],1,''ss'',
					[''no_more_child'',minfo.path_list,sig[3]]);
				|| (c[1] == nil) ->
					rnut_sstore([rnut_parent()],1,''ss'',
					[''no_more_child'',minfo.path_list,sig[3]]);
				|| true ->
					cinfo := searchpr.CHILDSINFO();
					rnut_sstore([rnut_parent()],1,''ss'',
					[''more_childs'',cinfo,sig[3]]);
				fi;
			fi;
		|| (sig[1] == ''exit'') -> rnut_kill(me,1);
		fi;
	od;
	';
```

_PVM

```
% scheme begin
var
	Host4: Host name='bream';
	Host3: Host name='manta';
	Host2: Host name='bonito';
	Host1: Host name='sole';
% scheme end
rel
	PVMSTART: -> out
	{c := self();
	for i := 1 to length(c) do
		if (c_getclass(c[i]) == 'Host') ->
			NutPrint(c[i]); host := c[i];
			rnut_addhosts(1,[host.name]);
		fi;
	od; };
```

_Append

```
var
	q, qq : array of any;
rel
	APPEND: q, in -> qq
		{ qq := [];
		qq := q;
		qq[Length(q)+1] := in; };
```

_Pool

```
var
	process_pool, free_mask : array of num;
	n_process, n_free_process : num;
rel
	MAKEPOOL : n_process, ws -> process_pool, n_free_process, free_mask
		{ process_pool := rnut_spawn(n_process,2,nil,'project/POOL.mem');
		rnut_pputwsr(process_pool,n_process,ws.childs_work_space);
		n_free_process := n_process;
		for i to length(process_pool) do
			free_mask[i] := 1;
		od; };
	STARTJOB : job,n_free_process,free_mask,n_process,process_pool -> n_free_process,free_mask
		{ for i to n_process do
```

```
                    if(free_mask[i] == 1) ->
                            free_mask[i] := 0;
                            sig := ['start',job,i];
                            rnut_sstore([process_pool[i]],1,'ss',sig);
                            n_free_process := n_free_process - 1;
                            exit ;
                    fi;
            od; };
        FREEPROCESS: id,n_free_process,free_mask -> n_free_process,free_mask
                { n_free_process := n_free_process + 1;
                free_mask[id] := 1; };
        EXIT : process_pool -> { rnut_kill(process_pool,length(process_pool)); };
```

_List

```
var
        mem : array of any;
vir
        in : any;
        out,arr : array of any;
rel
        CHECK : arr, mem -> out
                { out :=[];
                k := 1;
                for i :=1 to length(arr) do
                        found := false;
                        for j := 1 to length(mem) do
                                if (mem[j] == arr[i]) & (found == false) -> found := true; fi;
                        od;
                        if found /= true ->
                                APPEND(arr[i]);
                                NutPrint(k);
                                out[k] := arr[i];
                                NutPrint('aho');
                                k := k+1;
                        fi;
                od; };
        APPEND : in -> mem
                { if in /= nil -> mem[length(mem)+1] := in; fi; };
```

_Host

```
var
        name : text;
```

_ParSearch

```
var
        myinfo : NodeInfo;
        good_childs : array of any;
vir
        out : any;
        childsinfo : array of NodeInfo;
alias
        mypath = (myinfo.path_list);
        mycheck_list = (myinfo.check_list);
        mystate = (myinfo.my_state);
rel
        MAKEROOT : -> myinfo
                { myinfo.my_state := ROOT_STATE();
                mypath.APPEND(myinfo.my_state);
                mycheck_list.APPEND(myinfo.my_state); };
        APPENDPATH : -> myinfo { mypath.APPEND(myinfo.my_state); };
        ISGOOD : mystate -> out
                { if USRGOOD(mystate) -> out := true;
                || true -> out := false;
                fi; };
        MODIFY : myinfo -> good_childs
                { childs_state := USRMODIFY(myinfo.my_state);
```

```
                good_childs := mycheck_list.CHECK(childs_state); };
        CHILDSINFO : myinfo, good_childs -> childsinfo
                { for i to length(good_childs) do
                        childsinfo[i] := myinfo;
                        c <- childsinfo[i];
                        c.my_state := good_childs[i];
                od; };
```

_NodeInfo

```
var
        my_state : any;
        path_list : List;
        check_list : List;
```