GRADE User's Manual

G. Dózsa, T. Fadgyas, R. Lovas, P. Kacsuk

Contents

Introduction

GRADE is an integrated programming environment for development of parallel programs based on the message-passing programming paradigm. GRADE stands for GRaphical Application Development Environment and it consists of several software tools which assists the different steps of the development process. GRADE can be used to write, edit, execute and tune (debug, monitor, visualise) parallel programs.

The environment is implemented on the top of UNIX operating system and the X Window/MOTIF interface. It currently supports the PVM system as the target platform of the parallel applications, i.e. programs written in GRADE can be executed in any arbitrary PVM environment (e.g. in a UNIX based workstation cluster).

It contains a graphical program editor, called GRED (GRaphical EDitor), to construct the code of the parallel application. The programmer can define the parallel parts and activities of the program by using graphical symbols. The syntax and semantics of these graphical symbols are defined by the GRAPNEL language (GRAphical Process NEt LAnguage).

Additionally, the following tools are integrated into the GRADE environment:

- GRP2C: A precompiler to produce the C code of the graphically defined program.
- DDBG: A distributed debugger.
- PROVE: A performance visualisation tool.

As this manual put the focus on the user interface of the GRADE environment it will not give any information about technical details of the individual components.

Chapter 1

Installing the GRADE Package

1.1 Obtaining the Package

The source code of the GRADE package can be obtained in *tar*red and *gzip*ped form. The name of the file that must be down-loaded is **grade-XXX.tar.gz**[1]. The following command:

```
gzip -cd grade-XXX.tar.gz | tar xf -
```

will create the **grade-XXX** directory and will extract the source tree in it.

1.2 The Source Tree

The source tree of the GRADE package is organised as follows.

- **grade-XXX**: The root directory of the source tree.
  - **ddbg-XXX**: Source directory of the distributed debugger (DDBG)
  - **grp2c-XXX**: Source directory of the precompiler (GRP2C)
  - **gred-XXX**: Source directory of the graphical editor (GRED)
  - **prove-XXX**: Source directory of the performance visualisation tool (PROVE)

1.3 Software Requirements

As GRADE is intended to be used in a cluster of workstations to develop and execute parallel programs, it must be installed in different forms on the console host (*Console form*) and on the remote hosts (*Core form*).

1. *Console form*: it contains the user-interfaces of the different programming tools to drive the system (i.e. the "console").
2. *Core form*: it contains only those parts of the tools that GRADE needs on any host in the cluster to run properly but it does not contain any user interface.

1.3.1 Requirements to compile the system

It is highly recommended to use GNU C version 2.7.2 or higher to compile GRADE.

Furthermore, the following softwares must be installed:

- Requirements to compile in *Console form*:
  1. **Motif 1.2**
  2. **PVM 3.3.x** : PVM_ROOT and PVM_ARCH environment variables must be set properly.

[1] Use the actual version number in place of **XXX**.

3. **Tape/PVM 0.9** : TAPE_ROOT environment variable must be set properly and location of executables (e.g. "tapepp") must be listed in PATH.

- Requirements to compile in *Core form*:

1. **PVM 3.3.x**
2. **Tape/PVM 0.9** : TAPE_ROOT environment variable must be set properly and location of executables (e.g. "tapepp") must be listed in PATH.

1.3.2 Requirements to use the system

The following softwares must be installed so that GRADE can work (run) properly either in *Console* or in *Core form*:

1. **PVM 3.3.x**
2. **GNU Debugger (gdb)** version 4.16 or higher
3. **Tape/PVM 0.9** "tape_control" must be in a directory where PVM can find it to start (default location: $HOME/pvm3/bin/$PVM_ARCH)

1.4 Installing the System

To install GRADE in *Console form*:

1. Go to the root directory of the source tree (i.e. **grade-XXX**).
2. Type "**make CURDIR=$PWD**" to compile the whole system in *Console form*. The message "*GRADE has been compiled successfully !*" will appear on the screen if the compilation process is a success.
3. Type "**make install**" to install the compiled system in the directory "$HOME/grade" (which is the default value of the **GRADE_ROOT** variable defined in the Makefile). Having installed the system, the source tree can be deleted to save disk space.

To install GRADE in *Core form*:

1. Go to the root directory of the source tree (i.e. **grade-XXX**).
2. Type "**make core**". This command will compile the system and install it in the directory "$HOME/grade" in one step. ("$HOME/grade" is the default value of the **GRADE_ROOT** variable defined in the Makefile.) The message "*GRADE core stuff installed successfully!*" will appear on the screen if the compilation process is a success.

The value of the **GRADE_ROOT** must be the same on all hosts during the installation. For example, it can be "$HOME/grade" (which is the default) or "/usr/local/grade".

1.5 How to Start GRADE

To start GRADE means to start the user interface (i.e. the console) of the programming environment.

A shell script called "**grade**" can be found in the directory where the system has been installed in *Console form*. After setting a few environment variables this script starts GRADE. You can modify the value of some of those environment variables according to the information given in Sect. 1.6

1.6 Environment Variables

The following environment variables can be modified by the user in the "**grade**" startup script:

- **GRADE_ROOT** It is the directory in which GRADE has been installed actually. One can copy the installed system to another location by the folowing command:
  `cp -r actual-directory new-directory`
  After copying, the value of the **GRADE_ROOT** variable must be modified according to the new location of the system.

- **GRPDIR** It specifies the directory in which the GRAPNEL applications (i.e. the parallel programs written in GRADE) are saved. By default it is set to the directory :"**$GRADE_ROOT/examples**" which contains some example applications.

- **OBJDIR** It is a directory to store the object, cross-reference and other files that belong to GRAPNEL applications and used by GRADE.

- **BINDIR** It is the location of the executables of GRAPNEL applications. It must be a directory where PVM system can find user programs. The default is : "**$HOME/pvm3/bin/PVM_ARCH**". The **attacher** program, which is part of GRADE, must always be located in the **BINDIR** (i.e. **attacher** must be copied by the user to the new location if BINDIR has been changed[2]).

- **TEXTED** and **XTEXTED** They define the name of the texteditor which is used by GRADE to edit the textual parts of GRAPNEL programs. If the texteditor that is to be used has its own X Window interface then the **XTEXTED** variable must contain its name. For instance:
  `XTEXTED = xedit`
  Otherwise (i.e. the texteditor should be used in an xterm window), the **TEXTED** variable must be set, e.g.:
  `TEXTED = vi`
  Only one of them may be set at same time. The user should choose a texteditor which he is familiar in and available in his computer.

- **PVM_HOSTFILE** This variable contains the name of the PVM host file. Detailed information about the format of this file can be found in the PVM man pages (see 'man pvmd3').

[2] The program **attacher** can be found in the directory **$GRADE_ROOT/bin/$PVM_ARCH** too.

Chapter 2

The GRAPNEL Language

In GRADE, parallel programs can be constructed by the GRED graphical editor according to the syntax and semantics of GRAPNEL language.

GRAPNEL is a hybrid programming language in the sense that it uses both graphical and textual representation to describe the whole distributed application. Graphics are used to depict those program parts which are important concerning parallelism, but textual descriptions are applied where they are more appropriate (e.g. data declarations and definitions, code segments without any communication etc.). The main purpose of the graphical representation is to give a high level outline (or abstraction) of the distributed program where the key points are the communication events among the processes.

Here we give a brief description of the GRAPNEL programming model and the basic ideas behind the language. Chapter 3 describes the language elements supported by GRED currently[1]. To get a comprehensive description including all advanced features of GRAPNEL, the interested user should refer to [**?**].

2.1 The Underlying Programming Model

Our programming model is based on the message passing paradigm. The programmer can define processes which perform the computation independently in parallel, and interact only by sending and receiving messages among themselves.

In the current version of GRED, processes are always single units which can perform only point-to-point communication to interact and which are always created statically at the same time, i.e. when the whole parallel application starts.

Communication operations (i.e. sending and receiving data) always take place via communication ports which belong to processes and which are connected by channels. Every port of a process has its own protocol to ensure that the form of the transmitted data is the same at both the sender and receiver sides.

A communication channel connecting two processes denotes that those processes may transfer data at run-time. Inside the process the *input* and *output action* are used to perform the actual send or receive event.

In GRAPNEL programs, three hierarchical design levels are distinguished. At the top level the outline of the whole application is described graphically with respect to communication connections among the processes, while at lowest level the textual code fragments are given. (Remember, that GRAPNEL is a hybrid language.) At the middle level the send and receive events are defined inside the code of a process graphically. These levels are listed in top-down order as follows.

2.1.1 Application Level

This is the top level of the parallel application. Processes, communication ports and connections among the processes must be defined here graphically, but the functionalities (i.e. the code) of the processes are hidden at this level. In other words, processes are viewed as black boxes here and we are interested in only the messages transferred among them.

[1] Not all elements of the GRAPNEL language are supported by GRED currently but the editor is under development.

2.1.2 Process Level

This level is for describing the inner structure of a process graphically. (The inner structure means the control flow of the message transfer related parts of the process.) The point is, that the send and receive operations must appear graphically in the GRAPNEL code. This approach has two relevant advantages. Firstly, the programmers do not have to know the syntax of the underlying message-passing library, i.e. he can just put an icon into the control flow of the process and simple list the variables where the data is to come from (send) or where it is to be stored (receive), instead of having to write all the necessary PVM function calls[2] (e.g. different types of pack and unpack routines, buffer management, etc.). All low level message-passing system calls are generated by the GRP2C precompiler automatically based on the graphical information of the GRAPNEL code. Secondly, in the debugging and testing phase all message transfers can be animated in the graphical windows that can ease very much the identification of message-passing related bugs in the program. However, only those segments of the control flow must appear graphically which contain send or receive actions. Arbitrary large and sophisticated code fragment that does not contain any communication operation (i.e. that has nothing to do with parallelism) can be defined in ordinary textual way at a lower level (i.e. text code level) of the program.

2.1.3 Text Level

It is the lowest level of the GRAPNEL code. Here, the programmer can define textual C code fragments that representing the actual contents of the graphical icons defined at the process internal level. In the text code belonging to a textual block icon the programmer may call C functions that are defined either in library or in normal C files, thus C code written earlier can be reused easily.

[2]Actually, no PVM function calls are written in the GRAPNEL code at all.

Chapter 3

GRED: Constructing the Program Code

After starting, the programming enviroment is in editor mode as it is indicated in the center of the status bar at the bottom of the main window. Other tools can be started by choosing the proper items from the "Tools" pulldown menu.

Hint: All editor functions (except the select and open methods) are allowed only in editor mode.

In GRED, three different types of editor window – *Application Window*, *Process Window* and *Text Window* – represent the three hierarchical design levels of the GRAPNEL language (see Sect. 2.1).

We use top-down appproach to describe the editing mechanisms of different program elements, i.e. we begin with the elements of the *Application Window* and end with the text code fragments that belong to the different icons and can be edited in *Text Windows*.

3.1 Application Window

Fig. 3.1 depicts an *Application Window* and explains the different parts of it, including pulldown menus, toolbar, canvas, etc.

3.1.1 Open the Window

As the *Application Window* of GRED is the central consol of the whole GRADE environment, it appears on the screen automatically when GRADE is started by the user (see Sec. 1.5 for information about starting GRADE).

Currently, only one *Application Window* can be used during a GRADE session, i.e. the user cannot open a new window of this type.

3.1.2 Close the Window

The *Application Window* can be closed by choosing the "Exit" item from the "File" pulldown menu (see Sect. 3.1).

Warning: This window is the central console of GRADE, so closing it means to quit from the whole GRADE environment.

Hint: Do not forget to save the current application before quitting from GRADE.

3.1.3 Application (Language Element)

The application includes the whole code of the parallel program. It is identified with the "application name" defined by the user. Only one application can be edited by GRED at any time.

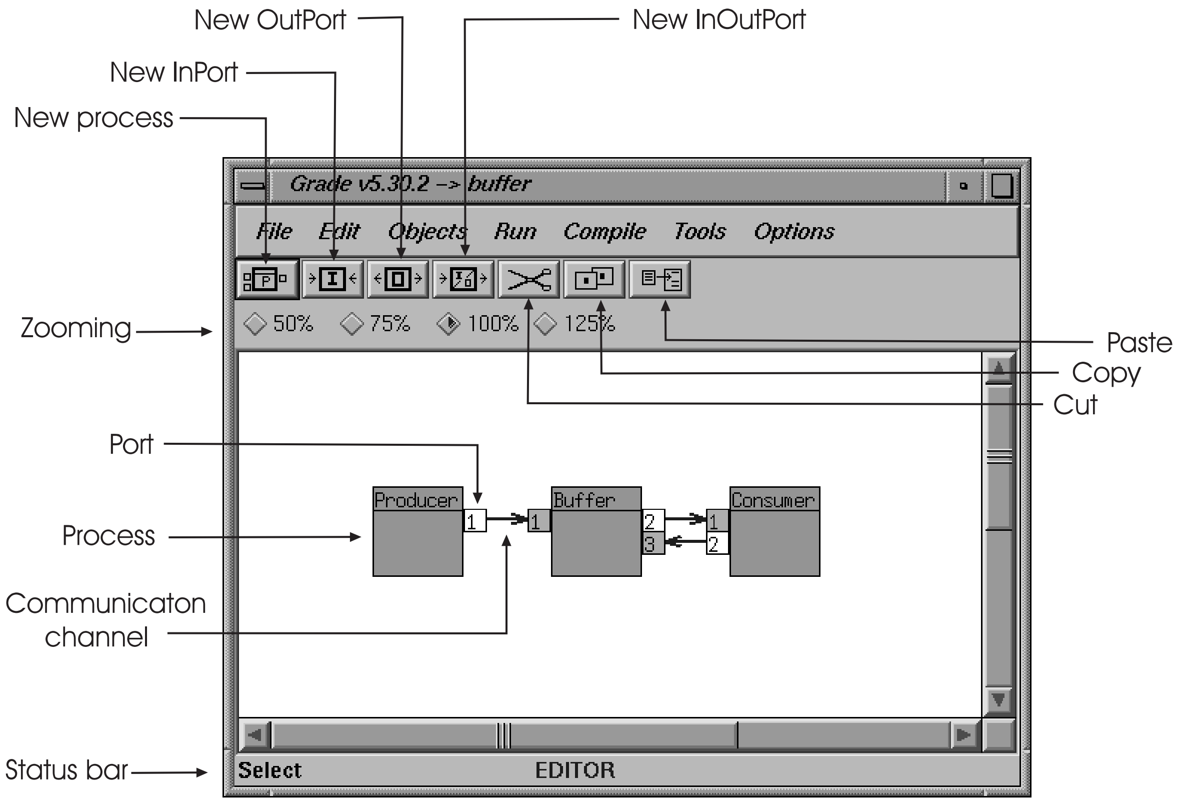

Figure 3.1: Application window

Create

A new application can be created by choosing the "New" item from the "File" pulldown menu (see Fig. 3.1). The name of it can be defined when the user save the application with "Save as" command(see Sect. 3.1.3).

Save

An application can be saved into a GRP file by choosing the "Save" item from the "File" pulldown menu(see Fig. 3.1). The default file name is "noname.grp". Other name can be defined by the help of the "Save As" menu item (see Sect. 3.1.3).

Save as

The name of the application can be changed by choosing the "Save as" item from the "File" pulldown menu. The user must define the name of the application in the dialog window that appears on the screen. GRED concatenates the ".grp" extension to the application name to get the name of the file. For instance, if the user defines the application name "prog1" then GRED saves the application into the file called "prog1.grp". Thus, application name must not contain any character that is not valid in a filename. The file is always created in the **GRPDIR** directory defined in the *grade* start-up script (see Sect. 1.5).

Open

An application can be restored from a GRP file by choosing the "Open" item from the "File" pulldown menu (see Fig. 3.1). The file to be open can be select in the file-chooser dialog appearing on the screen. By default, this file-chooser always shows the contents of the **GRPDIR** directory defined in the *grade* start-up script (see Sect. 1.5).

Hint: Do not forget to save the current application before open a new one.

Delete

Deleting the current application from GRED is equivalent with creating a new (empty) application (see Sect. 3.1.3).

3.1.4 Process (Language Element)

An application consists of processes (which can communicate with one another). Processes are represented by square icons on the canvas of the *Application Window* (see Fig. 3.1). Processes are identified by names defined by the user.

Create

A new process can be created in three steps:

1. Either choose the "Process" item from the "Objects" pulldon menu or push the "New Process" button in the toolbar (see Fig. 3.1).
2. Select the position of the new process icon on the canvas by moving the pointing device (i.e. mouse). A green box shows that the actual position of the icon is free. A red box with across lines can be seen if the position is reserved. Click when this position is appropriate for you.
3. Define the name of the new process in the dialog window apperaring on the screen.

The process name must not contain any character that is not valid in a filaname.(i.e. space, *, ...)

Select

The user can select a process icon by simple clicking on that icon. The frame of the icon is highlighted when it is selected. A selected icon can be deselected by clicking on the background. Only one process can be selected at any time.

Move

You can move process icons to new positions on the canvas in the following way (it works like a drag & drop method):

1. Select the process icon that you want to move with *middle* mouse button and press this button continously.
2. A green box shows the possible new position for a process according to the movements of the mouse.
3. When you are satisfied with the position of the green box, released the middle mouse button. It will cause the selected process to change its position to this place.

Rename

To rename a process, select it first and then choose the "Name" item from the "Edit" pulldown menu (see Fig. 3.1). Define the new name in the dialog window appearing on the screen.

Every process must have a unique name and process name may not contain any character that is not valid in a filename.

Copy-Paste

Copying a process requires the following steps:

1. Select the process icon (see Sect. 3.1.4).
2. Choose the "Copy" button in the toolbar (see Fig. 3.1). This will copy the process into an innner clipboard.
3. Either choose the "Paste" item from the "Edit" pulldown menu or push the "Paste" button in the toolbar. This will switch on the "process placing mode" (see Sec. 3.1.4).

4. Select the position of the new copy by clicking on the canvas.

The above method will copy the whole contents of the process, including communication ports (see Sect. 3.1.5), visual code (see Sect. 3.2) and textual code fragments (see Sect. 3.3). Only the communication channels (see Sect. 3.1.6) are not copied with the process.

Delete

To delete a process, select it first and then choose the "Cut" button in the toolbar (see Fig. 3.1).

Hint: As the "Cut" action places the process into the inner clipboard, a process being deleted accidentaly can be undeleted with a "Paste" action (see Sect. refproc-copy).

3.1.5 Communication Port (Language Element)

A process can send and receive messages via communication ports. Three different types of port are available currently:

- **InPort**: To receive messages from another process.
- **OutPort**: To send messegas to another process.
- **InOutPort**: To both send and receive messages to/from another process.

Ports are represented by small squares clung to the process that owns them and they are labelled by numbers for identification purposes (see Fig. 3.1).

Create

1. Select a process (see Sect 3.1.4).
2. Either choose the menu item "InPort", "OutPort", "InOutPort" from the "Objects" pulldown menu or push the equivalent toolbar button according to the type of the port you want to create (see Fig. 3.1).

For identification purposes, ports are labelled by numbers. Labels are generated automatically at creation time. Ports are placed around the process by the system.

Select

To select port, simple click on it. The border of the port is highlighted when it is selected. To unselect a port, click on the background. There only one port can be selected at the same time.

Move

Ports are always arranged around the process icon automatically. The user is not allowed to move them directly.

Delete

1. Select the port icon (see Sect. 3.1.5).
2. Choose "Cut" button in the toolbar (see Fig. 3.1).

Port Protocol

Every port must have its own protocol that determine the type and size of the messages to be transmitted via that port. The protocol can be defined as text code belonging to the port icon in the *Application Window* (see Sect. 3.3 for information about how to edit text code of an icon). The form of the protocol definition is the following:
`type1[size1]; type2[size2]; type3[size3]; ...`
It describes the structure of the message that can be sent or received via the port. The structure is defined as a sequence of arrays of different types.

Types can be used in the definition are the following[1]:

- **int**
- **char**
- **float**
- **double**

The size of the arrays can be defined by arbitrary C expressions which are evaluated every time when a message is transferred at run-time. Naturally, all variables and constants appearing in those expressions must be defined in either the "Global" or "Local" symbol or in a header file included in "Heads" symbol (see Sect. 3.2.3).

Example For instance, the user can define the text code belonging to a port icon in the *Application Window* as follows:

```
int[1]; char[s1]; double[MAX_SIZE+2];
```

assuming that varaible 's1' and 'MAX_SIZE' is defined in the text code of the process. This protocol means that the each message (i.e. byte sequence) that is sent or received via that port must be interpreted as the concatenation of one integer, some characters and some double values. The actual number of characters and doubles depend on the run-time value of 's1' and 'MAX_SIZE', respectively.

3.1.6 Communication Channel

Communication channels are applied to connect communication ports to define the end points of message transfers. They are represented by single lines with arrows(see Fig. 3.1). The arrows show the possible communication directions.

Two processes may commnunicate at run-time if and only if they have ports which are connected by communication channel.

According to the direction of the message flow, the following pair of ports can be joined by a channel:

- **InPort** and **OutPort**
- **InOutPort** and **InOutPort**

Channels define point-to-point communication between ports, i.e. each port can be connected at most one other port.

Create

1. Select the first port that you want to connect to another one with *middle* mouse button and press this button continously.
2. A rubber line shows the new communication channel according to the movements of the mouse.
3. When you reach with mouse pointer the other port, released the middle mouse button. It will cause the selected ports to joined to each other.

Hint: If you cannot create a channel between two particular ports then the types of the ports do not match probably (see Sect. 3.1.6 for the description of types which can be joined).

[1] The new version of the editor is to support arbitrary user defined types as well.

Select

A channel (i.e. the line) can be selected by simple click on it.

Delete

A channel can be removed in exactly the same way as the port or process. After its selecting, the "cut" button is used for the deleting the current channel.

3.1.7 Zooming Application Window

The size of the icons can be changed by the radio buttons in the toolbar.

3.2 Process Window

In a GRAPNEL application, every process has its own independent flow of control and address space. In C terminology, it means that each one has its own independent *main()* function. However, in case of GRAPNEL, these *main()* functions are generated by the **GRP2C** precompiler based on the visual code automatically[2]. **GRP2C** generates a separate C file containing the *main()* function for each GRAPNEL process. We will refer to this file as "process C file" in further sections.

In the *Process Window*, the user must construct the outline of this automatically generated *main()* function by using graphical symbols. The point is that every communication action (e.g. send, receive) must be defined by an approprite visual program element. However, arbitrary large textual C code segment that contain no communication with other processes can be defined by a single icon (see Sect. 3.2.4).

Fig. 3.2 depicts a *Process Window* and explains its static parts (i.e. those parts that are created by GRED automatically in case of every *Process Window*).

3.2.1 Open the Window

To open a *Process Window* for a process:

- Select the process (see Sect. 3.1.4)
- Choose the "Open" button from the toolbar.

You can also open a process window a double mouse click on the proper process.

3.2.2 Close the Window

Choose the "Close" button from the button bar (see Fig. 3.2).

3.2.3 Data Declarations and Definitions

The user can define/declare data that that can be used in the C code belonging to the visual instrctions. There are three static icons for this purpose in every *Process Window*: "Global", "Local" and "Heads" (see Fig. 3.2). They are explained as follows.

"Global": Global Data

The "Global" static icon is represented by a box labeled as "Global" at the top of the canvas (see Fig. 3.2). The programmer can define/declare global variables, types or constants according to the C syntax as text code belonging to this symbol. See Sect. 3.3 for information about how to edit text code of an icon.

The term "Global" means that data defined/declared here can be referred in every piece of text code belonging to the particular process, even if it is in a completely separate C file.

[2] Of course, other C functions needed by a process or the application can be defined by the user in ordinary C files. (See Sect. 4 for information about how to use external C files or libraries)

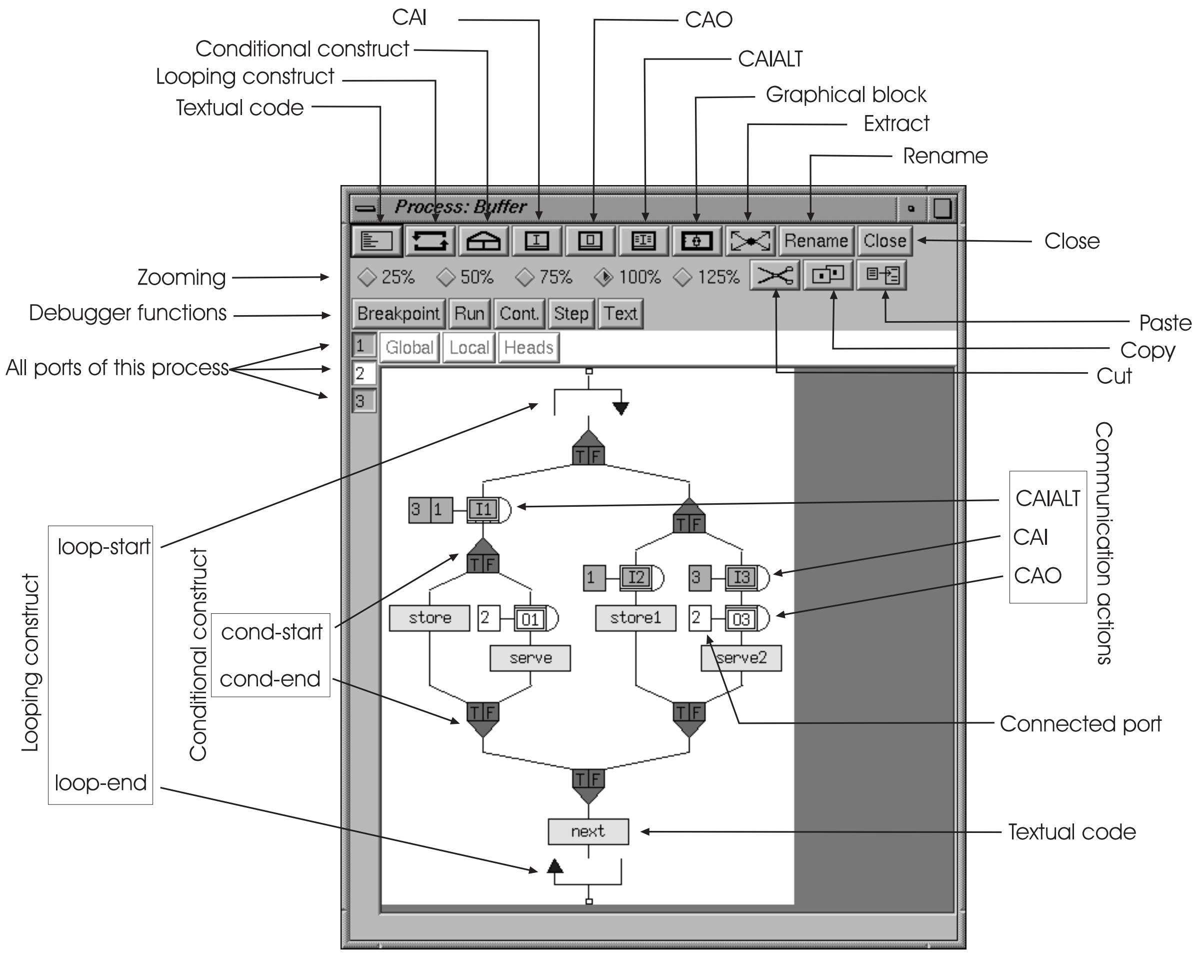

Figure 3.2: Process window

The C code defined for "Global" is put by GRP2C in front of the definition of the *main()* function in the "process C file" (see Sect. 3.2).

To define the actual C code click on the icon.

Example For instance, one can define the code for "Global" as follows:

```
#define BOOLEAN int
int  i1;
static int i2;
extern int i3;
```

As a result, the **GRP2C** precompiler will produce a "process C file" something like this:

```
/* ******* GRP2C: C file for Process X ******* */

⋮

#define BOOLEAN int
int  i1;
static int i2;
extern int i3;

int main()
{

⋮
```

```
}
/* ***************************************** */
```

"Local": Local Data

The "Local" static icon is is represented by a box labelled as "Local" at the top of the canvas (see Fig. 3.2).

The programmer can define local data in the text code onwed by this symbol. Local means, that the scope of these variables is restricted to the text code fragments belonging of the visual instructions defined of the particular process. (They can not be accessed in separate C files.)

The GRP2C precompiler puts the text code belonging to the "Local" icon at the very begining of the definition of the *main()* function in the "process C file" (see Sect. 3.2). In other words, the local variables and constants of the *main()* can be defined in this icon.

To define the actual C code click on the icon.

Example For instance, one can define the code for "Local" as follows:

```
int  i,k;
static int i2;
double d;
```

As a result, the **GRP2C** proecompiler will produce a "process C file" something like this:

```
/* ******* GRP2C: C file for Process X ******* */
⋮

int main()
{
 int  i.k;
 static int i2;
 double d;

⋮

}
/* ***************************************** */
```

"Heads": Include Files

The "Heads" static icon is located beside the "Global" and "Local" symbols (see Fig. 3.2). The user can define C "#include" directives as text code belonging to it in order to use predefined C code or data located either in libraries or in separate C files.

GRP2C puts the text code fragment owned by this icon at the very begining of the "process C file" (see Sect. 3.2).

To define the actual C code click on the icon.

Example For example, the user can define the following code for the "Heads" symbol (see Sect. 3.3):

```
#include <string.h>
#include "myheader.h"
```

As a result, the **GRP2C** precompiler will produce a "process C file" something like this:

```
/* ******* GRP2C: C file for Process X ******* */
#include <string.h>
#include "myheader.h"

⋮
```

```
int main()
{
  ⋮
}
/* **************************************** */
```

3.2.4 Visual Instructions in the Process Window

GRED provides several visual instructions (i.e. icons) for the user to define the outline of the *main()* function of a process graphically. The point is that every communication operation (i.e. send and receive) must be defined by an appropriate graphical symbol. Text code fragments in which no commnucation appear can be defined by a single icon in this graphical control flow.

We explain the available visual instructions as follows.

CAO, CAI, CAIALT: Communication Actions

Communication Actions (CA) are used to send/receive some data to/from other processes. CA icons must be connected to one or more port symbols (see Sect. 3.2.7) to define the source/target of the message transfer. Three different types of CA are provided by GRED:

- **CAO** : send a message via every connected port. Different types of messages can be sent via different ports. The process blocks until each message is sent.
- **CAI** : receive a message via every connected port.Different types of messages can be received via different ports. The process blocks until a message is received via each port.
- **CAIALT** : select only one message to receive via the connected ports. The process blocks until a message is available via one of the ports. If more than one messages are available, the selection is non-deterministic.

The text code belonging to a CA icon must be a colon seperated list with the names of the variables where the data to be transferred should be stored (CAI, CAIALT) or should come from (CAO). If more than one port are connected with the icon then every port must have its own list in the text code and the port must be identified in front of the list (see examples below).The type and size of the variables in each list and the protocol of the corresponding port must match strictly. See Sect. 3.1.5 for information about how to define the protocol of a port. Furthermore, a guard condition can be defined for each port in the text code, i.e. the message transfer is performed only if the condition is evaluated to true at run-time.

For example, if the CA icon is connected only with one port symbol the user can define that text code belonging to that icon as:

```
kalap; sapka;  csakany;
```

assuming that these variables are defined in either the "Global" or the "Local" symbol (or in a header file included in "Heads") and the types of them match with the protocol defined for the port. If the CA icon is connected with more than one ports (e.g. with port number 2 and number 4), each port must be identified in fron of the list of variables:

```
PORT 2:
   kalap;
PORT 4: GUARD ( ! full && i <= 100 ) :
   csakany; sapka;
```

As it can be seen, a guard condition is defined for the communication to take place via port number 4. The condition can be any valid C expression. 'PORT' and 'GUARD' are reserved words.

SEQ: Textual Block

The SEQ (SEQuential) visual instruction (icon) can be used to define an arbitrary large text code fragment that does not contain any communication operation. SEQ icon is labelled by a name defined by the user at creation time. The text code belonging to the icon can be defined as it is described in Sect. 3.3. The code may contain references to external functions or data (see Sect. ?? for more information about how to use seperate C files and libraries).

GRPH: Graphical Block

The GRPH visual instruction is provided to support the structured design of the graphical control flow of a process. Its appearance is similar to that of the Textual Block (see above) but the contents of it is defined by using visual instructions instead of text code. As a result, it may contain communication opretaions (CAI, CAO or CAIALT icons). In fact, it may contain any kind of visual instructions, so GRPH icons can be nested. The contents of the Graphical Block can be edited after its selecting by clicking on the "Open" button. Fig. 3.3 depicts a GRPH icon and *Graphical Block Window* being opened for that block. The apperance and functionality of this window is exactly the same as that of *Process Window* except that the static symbols for data declarations/definitions (i.e. "Local", "Global" and "Heads") miss here.

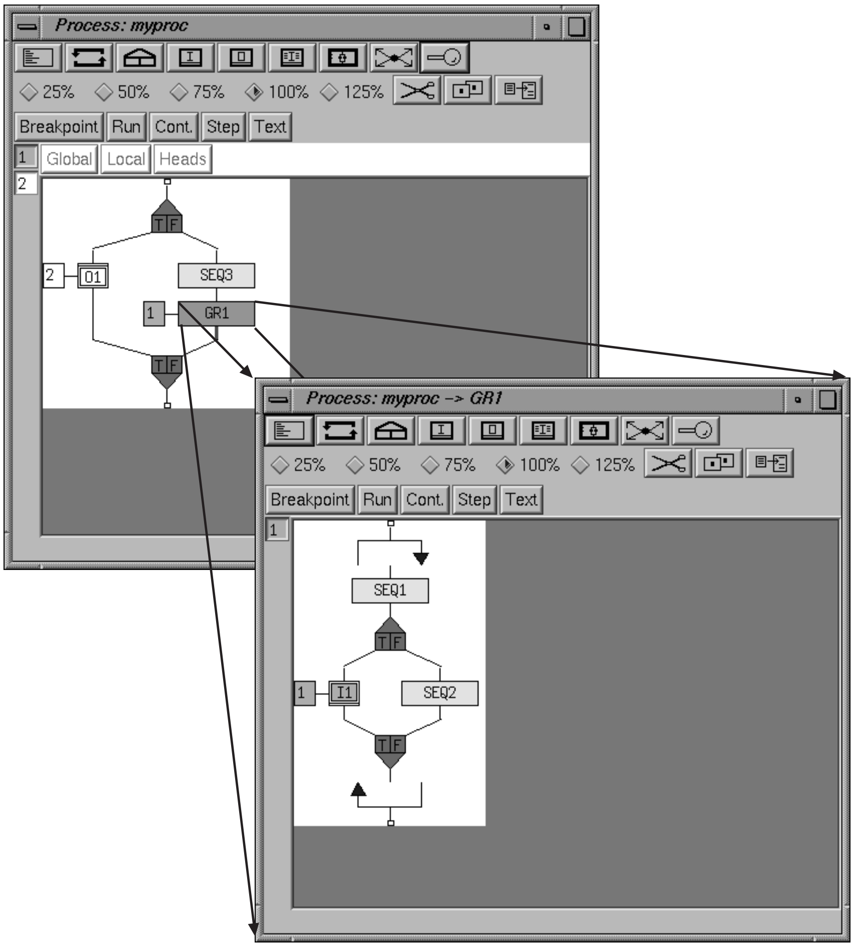

Figure 3.3: Graphical block

A GRPH icon can be connected with ports (see Sec. 3.2.7) since communication operations may need ports inside GRPH. As a result, GRPH icons can be viewed as complex communication instructions.

LOOP: Looping Construct

A visual LOOP instruction consists of two icons (loop-start and loop-end, see Fig. 3.2) that wrap around a part of the graphical control flow (i.e. wrap around the body of the cycle). The user must apply visual LOOP instruction when communication with other process(es) is needed inside a loop. The type of the loop is determined by the text code belonging to the loop-start and loop-end icons any can be any usual C loop construct, i.e. "for", "while" or "do". (See Sect. 3.3 for information about how to define text code for icons.) In case of "for" and "while" the loop-end icon must be empty (i.e. no text code can belong to it) and in case "do" the loop-start icon must not contain any text code.

Example For example, to set up a visual "for" loop the user can define the following text code for the loop-start icon:

```
for(i=0; i< MAX; i++)
```

assuming that 'i' and 'MAX' are defined in either the "Global" or the "Local" symbols, or in an include file in "Heads".

COND: Conditinal Construct

A visual COND instruction consists of two icons (cond-start and cond-end, see Fig. 3.2) that show the start and end point of the alternative pathes of the grahical control flow. In other words, COND is a visual 'if' instruction. The user must apply visual 'if' instruction when communication with other process(es) is needed on one particular path of the control flow. The condition must be defined as text code belonging to the cond-start icon. (See Sect. 3.3 for information about how to define text code for an icon). The "True" and "False" graphical branches ar labelled as "T" and "F", respectively, by GRED.

Example For example, the user can define the text code of the cond-start icon as:

```
if (alma < 10)
```

assuming that 'alma' is defined in either the "Global" or the "Local" symbols, or in an include file in "Heads".

3.2.5 Creating Visual Instruction

All types of visual instruction can be inserted in the same way into the graphical control flow:

1. Select the control line segment that you want to be split by clicking on it.
2. Choose the approprite button – SEQ, GRPH, CAI, CAO, CAIALT, COND, LOOP – in the toolbar (see Fig. 3.2).

In case of COND and LOOP, the start and end icons are always inserted together to ensure that only well structured control flows can be created by the user.

Furthermore, you can create a graphical block from a group of selected visual instructions by choosing the "graphical block" button when the selection is active. A graphical block can be extracted in place by the "Extract" button (see Fig. 3.2).

3.2.6 Select a Visual Instruction or a Block

You can select an icon a single click on it. In case of the select COND or LOOP structure, all of the icons are selected inside them.

For expanding the selected area you have to click with the middle mouse button on a visual instruction to define the top or bottom border of the selection.

3.2.7 Connection between Visual Instructions and Ports

All ports of a process are represented by port icons at the edge of the *Process Window* (see Sect. 3.1.5 for information about how to create ports for a process). Visual instructions that define communication events must be connected with the appropriate port icons in the *Process Window* to define the the source or the target of that send or receive operations.

There are some rules that restrict the possible connections between icons and port symbols, namely:

- A CAI icon (see Sect. 3.2.4) must be connected to one or more "InPort" or "InOutPort"symbols.
- A CAO icon (see Sect. 3.2.4) must be connected to one or more "OutPort" or "InOutPort" symbols.
- A CAIALT icon (see Sect. 3.2.4) can be connected to one or more "InPort" or "InOutPort" symbols according to the text code belonging to that particular CAIALT.
- A GRPH icon (see Sect. 3.2.4) can be connected with ports without any restriction.
- Only CAI, CAO, CAIALT and GRPH icons can be joined with port symbols.

Connecting an Icon with a Port To connect an icon with a port symbol select the icon and the port symbol to be joined by clicking on them in succession.

Remove Connection between an Icon and a Port A channel can be removed in exactly the same way as it has been created.

3.2.8 Copy Visual Instruction(s)

You can put the selected visual instruction(s) into the clipboard by the "Copy" button (see Fig. 3.2). The "Paste" button can be used for inserting the contents of the clipbord to the selected control line segment at any time .

3.2.9 Delete Visual Instruction(s)

To delete visual instruction(s):

1. Select the icon(s).
2. Choose the "Cut" button (see Fig. 3.2).

Visual instructions consisting of two icons – LOOP and COND, see Sect. 3.2.4 and 3.2.4, respectively – are deleted together with their contents.

3.2.10 Rename Visual Instruction

To rename an icon, select it and click on the "Rename" button (see Fig. 3.2).

3.2.11 Zooming Process Window

The size of the icons can be changed by the radio buttons in the toolbar. If you choose the 25% size, the names of the icons are invisible.

Warning: The size of the window is changed automatically. It means that this method tries to minimize empty place around the diagram.

3.2.12 Notes

All the available functions can be performed from the right button popup menu.

3.3 Text Window

The textual code belonging to an icon can always be edited by double clicking on it. This causes an externel texeditor to be appeared on the screen in a new window. The type of this editor can be set in the *grade* startup script (see Sect 1.5).

The text code fragments are passed between GRED and the external texteditor via temporary files created and destroyed automatically by the system. This fact imposes two important rules on the user:

1. ALWAYS SAVE THE TEXT after modifying it in the external texteditor.
2. NEVER CHANGE THE NAME of the file in the external texteditor.

GRED does not allow a text code fragment to be edited by more than one texteditor at the same time.

Hint: If you try open a text window of an icon and nothing happens then the text code belonging to that icon is probably already being edited in an external texteditor. Find that texteditor instance on the screen.

Chapter 4

Compiling the Code

As GRADE has been intended to work in heterogenous cluster of workstations, the executables of user processes must be produced on each host in the cluster since the hardware architecture may differ significantly and mapping decisions (i.e. which process runs on which processor) can vary from run to run of the application.

The short term "compilation" is used in further sections to denote all activities together required to produce the executables from the visual GRAPNEL code.

The user can compile the visual code on any host in the cluster with the help of the "Compile" pulldown menu in the *Application window* (see Fig. 3.1). This menu contains the name of each host in the user's PVM cluster.

GRADE reads the contents of the *PVM hostfile* when it starts in order to know the name of the hosts comprising the PVM cluster of the user[1]. The location and the name of this file is defined by the **PVM_HOSTFILE** UNIX environment variable in the *grade* start-up script (see Sect. 1.5 for more information).

4.1 Compiling on Localhost

To compile the GRAPNEL code on the localhost (i.e. on which the GRADE console is running) choose the "Localhost" item from the "Compile" pulldown menu (see Fig. 3.1). Having chosen it, the *Compilation Window* appears on the screen to show all information about the compilation process. Section 4.3 describes the appearance and functionalities of that window.

4.2 Compiling on Remote Hosts

To compile the visual code on a particular remote host, choose the name of that host from the "Compile" pulldown menu (see Fig. 3.1). Having chosen it, the *Compilation Window* appears on the screen to show all information about the compilation process. Section 4.3 describes the appearance and functionalities of that window.

To compile the GRAPNEL code on all remote hosts simultaneously, choose the "All Hosts" from the "Compile" pulldown menu (see Fig. 3.1). Having chosen it, several instances of *Compilation Window* appear on the screen – one for each remote host – to show information about the compilation process on the different hosts. Section 4.3 describes the appearance and functionalities of that window.

Hint If the name of a remote hosts does not appear in the "Compile" pulldown menu then either the *PVM hostfile* does not contain that host or GRADE does not find the file itself probably.

4.3 Compilation Window

This window shows all information related to the compilation process on a particular host. Figure 4.1 depicts a sample instance of *Compilation Window*. It has the following areas:

[1] The syntax of the file is described on the manual pages of PVM (i.e. try "man pvmd3").

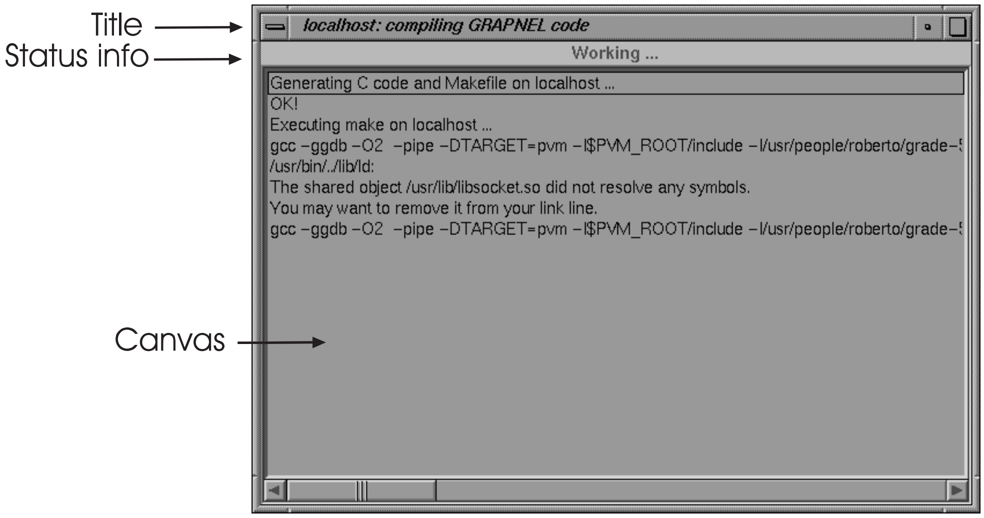

Figure 4.1: Compilation window

- **Title** It contains the name of the host on which the visual code is compiled.
- **Status Info** It shows the current status of the compilation process which can be:
  1. *Working*: the compilation is still in progress.
  2. *Succes*: the compilation has finished successfully, i.e. the executable code on that particular host is ready to run.
  3. *Failure*: the compilation failed because of some errors. Error messages are shown on the canvas. See Sect. 4.3.1 for information about how to find errors in the visual code.
- **Canvas** It contains all necessary information concerning the compilation process in a scrolled list form. The list is extended continuosly as the compilation proceeds.

In order to help in understanding the information appear on the canvas, we give a brief description of the actions that take place in succession during the compilation and that of the most common reasons of failures.

1. Copying the GRP file to the (remote) host. The GRP file is always stored on the console host, thus, in case of a remote host, it must be copied to that host. Possible reasons of failure:
   - GRADE is not installed on the remote host in *Core form* (see Sect. 1.4 for information about installation of GRADE).
   - The user has different login name on the remote host[2].
2. Invoking the GRP2C precompiler to produce the C files and Makefile from the GRP file. Possible reasons of failure:
   - There is not GRP file at all as the application has not been saved yet.
   - There is (at least) one incorrect part in the visual code (e.g. A port is not connected anywhere or the protocol of a port is missing). Next section gives some information about how to find errors in the visual code.
3. Invoking the UNIX *make* utility to produce the executables from the C code.

   Possible reason of errors:
   - Incorrect C code fragment is defined for one of the visual instructions. See next section for information about how to find errors in the visual code.

[2] GRADE invokes the "rcp" UNIX command which requires the same login name on the remote host.

The following directories are used by GRADE to store the files related to GRAPNEL applications:

Directory	**Files stored**
GRPDIR	GRP files
OBJDIR	generated C files, object files, makefile
BINDIR	executables

Values of the environment variables **GRPDIR**, **OBJDIR** and **BINDIR** can be changed in the *grade* start-up script (see Sect. 1.5).

If errors occur during the compilation the error messages are shown, otherwise, the "OK" message appear at the end of each above listed action.

4.3.1 Error Fixing in the Visual Code

If an error is caused by some problem in the GRAPNEL code and the user click on the error message in the *Compilation Window* then the error related parts of the visual code are highlighted by the system to assist the user in finding the location of the incorrect code.

For example, if the visual code of the process "myproc" contains a SEQ icon in which an undefined variable "alma" is used then the user will get an error message in the *Compilation Window* something like this:

```
myproc.c: In function 'main':
myproc.c:89: 'alma' undeclared (first use this function)
myproc.c:89: (Each undeclared identifier is reported only once
```

These messages are produced by the GNU C compiler during the compilation of the `myproc.c` "process C file". (The "process C file" is generated by the GRP2C precompiler from the visual code automatically, see Sect. 3.2). The user can click on the above messages in order to make GRADE highlight the process icon and visual instruction that contains the undefined variable "alma".

Chapter 5

Get the Program Running

Having compiled the visual code on every hosts of the user's PVM cluster (see Chapter 4), a GRAPNEL application can be executed either by the help of the GRADE console or as a normal PVM program without using GRADE. In both case, the point is that the so called *Application Server* process must be started. This process is always located in the "OBJDIR" directory defined in the *grade* start-up script (see Sect. 1.5). The name of the *Application Server* process is the same as that of the user's application. For instance, if the application is saved into a GRP file called "myapp.grp" the name of the *Application Server* will be "myapp".

5.1 Execute the Application from GRADE

1. choose the "Start PVM" item from the "Run" pulldown menu in the *Application Window* if PVM has not started yet.
2. choose the "Run All" item from the same pulldown menu.

5.2 Execute the Application without GRADE

1. start the PVM deamon (if it has not started yet)
2. start the *Application Server* process with the "-n" option.

Chapter 6

Debugger

To start the debugger integrated into GRADE, choose the 'Debugger' item from the 'Tools' pulldown menu. In the main window, the process icons are coloured lightblue as soon as the debugger is ready to work (it usually takes for a while, so be patient). The bottom row of the toolbar in the process windows contains buttons for creating breakpoints ("Breakpoint"), starting the process under the control of the debugger ("Run"), making the process continue its execution if it is stopped ("Cont.") or making the process go ahead one visual instruction ("Step").

The last button ("Text") can be used to open a text debugger console for a process in order to debug the C code fragments belonging to the visual instructions. Breakpoints in the C code can be defined by simply clicking on a line of the code. Values of program variables can be examined in this console as well.

You are able to put a breakpoint on an icon (in a process window) by the help of "breakpoint" button after selecting that icon. The breakpoint is represented by a circle inside the icon. To delete a breakpoint you have to act similarly to its creation.

Chapter 7

Prove

To visualize the runtime behaviour of your program, you should do the following steps:

1. Switch on the "Generate trace file" flag in the "Compiler options" window ("Options" pulldown menu).
2. Recompile your program ("Compile" menu).
3. Execute the application ("Run all" from "Run" menu).
4. After the application has finished, start PROVE ("Prove" item from "Tools" pulldown menu).

7.1 Space-time diagram

The main window of PROVE depicts a space-time diagram. Time axis is represented at the bottom of the window and activities of processes are depicted as thick horizontal lines above the time axis. Different colors correspond different kinds of activities. Arrows between the thick horizontal lines represent message transfer between processes. Communication ports, which the message was transferred through, are represented at both processes.

7.2 Zooming

You can zoom in any part of the figure by using "rubber band" technique, i.e. select a rectangle area while holding down the left mouse button. You can perform zooming operations in successions and you can always step back to the previous (i.e. not zoomed) figure by clicking with the right mouse button.

7.3 Relation between events and the visual code of the application

You can find the visual instructions in the code of processes which were active at a certain point of time in the following way:

1. Click with the middle mouse button on the time axis while holding down the "Shift" key. A vertical dashed line will appear in the figure.
2. Click – near the dashed horizontal line – with the middle mouse button on the thick horizontal line of any process. The icon belonging to that process and the visual instruction being active in the process at that point of time will be highlighted in the graphical editor windows of GRADE.

Similarly, you can find the next point of time when a particular visual instruction was executed by clicking – with the middle mouse button – on that instructions in the editor window. The dashed vertical line in the PROVE space-time diagram will move ahead to show the next moment when that instruction was executed.

7.4 Further diagrams of PROVE

1. Gantt chart : select "Process State" from "Windows" pulldown menu to open.
2. Volume of the data transferred between pair of processes : select "Process Communication" from "Windows" pulldown menu to open.
3. Volume of data transferred between pair of hosts : select "Host Communication" from "Windows" pulldown menu to open.

You can sycnhronize all the above mentioned diagrams with the space-time diagram (i.e. with the main window of PROVE) if you select the "Full/Prop" item from the "Display" pulldown menus. Synchronisation means that all windows will show exactly the same time interval which is selected by zooming and scrolling in the main window.

7.5 Filtering the information displayed

You can filter out unrelevant information from the diagrams by the help of the filter dialog window. To open this window, choose "Filter Dialog" from the "Information" pulldown menu in the main window.

Chapter 8

Frequently asked questions

This chapter is recommended for users who are *not* experienced with the GRADE.

1. "How does the system **generate the names** of copied or new processes ?"
   An increased number is concatenated to the original process name in all case.

2. "Why does not appear any **textual window** if I push "Open" button?"
   If the XTEXTED or TEXTED environment variables (see Sect. 1.6) are set wrong, the system will not warn about it. So check it carefully.

3. "How can I **delete** anything in GRED ? I can not see anywhere "Delete" button..."
   The main concept is using "Cut" button instead of "Delete" button so you will be able to restore the cut parts.(like an UNDO method)

4. "Can I write **C++** code into the textual block ?"
   It is not supported yet.

5. "I have launched the **PROVE** and the diagram seems to be **wrong**."
   You have to set on the "Generate trace file" check box in the "Compiler options" window, than recompile your application and run it again.

6. "I would like to **get** some **input data** from the user in my program but the standard scanf *C* function does not work."
   You have to use the grp_scanf function in this case. **Important !** Its usage is *not* the same as the original function. Only one value getting is allowed in one function and you must define its type with a separate argument.
   Example:

   ```
   grp_scanf("N factor :",&Nfactor, "DOUBLE");
   ```

   The following types are valid: *BYTE, INT, FLOAT, DOUBLE, LONG, STR*

7. "I would like to **show** some **information** in the application console window but the printf *C* function does not work."
   The grp_printf funcion is used for printing on the application console. The arguments are the same as native printf function.
   Examples:

   ```
   grp_printf("Producer sent %d: %f to buffer. \n", i,d);
   ```

8. "Is the asynchronised send and receive supported by the system ?"
   Only the synchronised transfer was implemented in the GRADE.

9. "How can the system forget the **changes in the text code segment** when I have saved it surely by the text editor ?"
   If the system is not in editor mode, the changes will be dropped automatically. (e.g. it is nonsense to change source code or process topology during a debugging) Select the editor tool in *Tools* pull-down menu and you can edit the text code again.

10. "How can I set the **colors** ?"
It can be reached by editing the *main.rc* resource file. The changes can be seen after a restarting GRADE. Example:

```
AppConsol*background :lightgray
BuildExe*background  :lightgray
Debuger*background   :lightgray
Prove*background     :lightgray
```

The valid colors can be found in all *X* reference books. Note: Most of the colors are fixed by the code as yet.

11. "What is the **right sequence** of variables in the CAI or CAIALT boxes ? I would like to **transmit** 3 integer in a given sequence: a, b, c... "
The sequences of variables must be the same in the input and the connected output communication actions. It works like a FIFO.

12. "Should I know programming with **PVM** to use GRADE ?"
GRADE hides all the low-level PVM functions so the PVM knowledge is not important.

13. "I have planned a complex process topology with **many connections** and I can not see which ports are connected to each other. What can I do in this case ?"
The selected communication channel is painted by red color, just click on the channel and the related ports become visible.

14. "I have deleted the contents of **/tmp directory** and the GRADE does not work correctly... What is the problem?"
You can find some files in the */tmp* directory which is responsible for the communication among the GRADE tools while the system is running. If its contents is deleted, the GRADE does not work correctly.

15. "Where can I find some information about the error after a system crash or a bug?"
In the */tmp* directory you can find a file *grade.out.xxx* where the last 3 or 4 digits show your user ID. This file may includes some details about the reasons of the error.

16. "Can I change the **port protocol** in run-time ?"
The new version GRADE supports the change of port protocol. You can define the array size with an expression which can include global and local variables, constants and arithmetic operations. *Important!* If you want to take advantages of this new feature, you must use it very carefully. When the input and output side protocols become different, your application may collapse.

17. "Where can I set the target **hosts**?"
A simple mapping tool is provided to set hosts in GRADE system. The *Mapping* item in the *Tools* pull-down menu opens a text editor window in which you can define where the system has to run the processes. Example:

```
Producer        venus.lpds.sztaki.hu
Consumer        jupiter.lpds.sztaki.hu
Buffer          mars.lpds.sztaki.hu
```

Chapter 9

Fixed bugs and new features

This chapter is recommended for users who are experienced with the GRADE.

9.1 Generally about the changes

The cut-copy-paste mechanism is the main new feature of the system, it can be accessed in all windows. For the more correct displaying, the zooming steps are implemented in some discrete steps. "Delete" buttons do not exist further.

9.2 Application window

The window have become resizeable and scrollable so the 100 processes limit does not exist.

9.3 Process window

The way of select was improved with block and fast structure select methods. If it is necessary, the graphical block can be extracted. A popup menu, which activated by the right mouse button, contents of the all valid operations.