COLOMBO

PROJECT COLOMBO
(Project No.: 24,907)

REPORT D9:
CAMELot 1.3 Implementation and User Guide

(AVAILABILITY: Public)

Workpackage: 3

Task: 3.6

Authors: K. Kavoussanakis, S. D. Telford, S. Booth, L. Clarke, A. Smith, A. Trew, A. Simpson, G. Spezzano, D. Talia.

Date of issue: 29 September 2000

d9.doc

Table of contents

1.   Executive Summary

1.1 Objectives

Workpackage 3 of the COLOMBO project aimed to produce a software tool for the programming and execution of Cellular Automata on parallel computing platforms. The tool, named CAMELot, was used within the Colombo framework for the programming and execution of bioremediation simulation software. Although this defines the current end-users of the software and therefore influences largely the design, the final product is not bioremediation-specific.

It was important for the software to be available on state of the art parallel computing platforms, including Massively Parallel Platforms, like the Cray T3E and the Quadrics CS-2; Shared Memory Platforms, like the Sun Enterprise-3500; and closely or loosely coupled clusters of workstations or PCs, like Beowulf clusters or any local cluster of workstations in a typical networked modern office environment. The scope of the underlying operating systems was limited to Unix variants and that of the Graphical User Interface (GUI) to X-Windows, although it would be desirable to develop the software with a view to allow further portability.

CAMELot was to be more user-friendly than its predecessor both in the user interface and the associated programming language, CARPET. It was intended that the internals of the underlying program which supports the parallel execution of the CARPET program be hidden from the user. It was also desired that parallel execution be straightforward to apply and unrelated to the CARPET program. After these basic requirements were satisfied within the predefined schedule, the onus fell on the requirements of the end-users (CRA, ENEA and ISI-CNR) which were only aiming towards a general CA programming tool. It was also anticipated that the performance of the system would be adequate in order to exploit the enormous computational power of the intended architecture. Performance optimisations were sought throughout the development of the tool.

1.2 Tools and Methods

The CA program was developed in C, a widely available programming language. Parallelism was achieved by means of the MPI message passing standard. This provides support for parallel execution without needing to incorporate the details of the processing elements to the code. MPI implementations are also available in all the desired computer platforms.

The graphical user interface was developed using X-Designer, a graphical tool which can generate X-Windows code. The tool can also reuse the native code it has generated to produce MS-Windows code thus enabling portability further than the initial requirements. It would be wrong to infer that porting to MS-Windows is straightforward though. X-Designer only provides the windows; the functionality behind them has to be developed separately and coded manually.

A similar GUI look-and-feel to the predecessor tool was intended, as no serious drawbacks were identified. The users were invited to contribute ideas and share their views about the quality of the software by means of a system which was also able to record the progress of the interactions between end users and developers. This was not activated until the state of the tool had been stabilised. Alpha and beta releases of the software were however made available to the users according to the predefined plan [Clarke et al. 1998].

The above suggest an iterative software development method, namely evolutionary delivery, by which the architecture is defined initially and then a develop-deliver-feedback-enhancements cycle is followed. This method has the advantage that it gives good visibility to the project's development and is flexible enough to produce good results even if the original time and effort estimates prove to be inaccurate. The weakness of the model is the idleness of the developers during the feedback solicitation period; this was handled by exploiting the time to develop the report documenting the use and implementation of the system, a version of which you are reading now.

The method for feedback is also specified in [Clarke et al. 1998]. EPCC implemented at a specified stage (Workpackage 3.5) a system through which the users were able to report bugs and submit enhancement requests. EPCC ran the system and assigned the reports to the appropriate partner who reported back to the system about the progress of these reports. This enabled tracking and archiving of the progress of the project after the point where the initial design [Telford et al. 1998] stopped.

1.3 Results

The Workpackage produced three interrelated components:

- A Graphical User Interface for the development of CA code;
- A CA Engine to enable the parallel execution of the CA code to parallel computers;
- A new version of the CARPET CA programming language.

The software is available in all the architectures originally targeted. It eliminates weaknesses of its predecessor, both in the user interface and CARPET. Finally, the vast majority of the features requested by the users were implemented. The bioremediation simulation code developers agree that the open issues identified in section 10 of this document were not important for the fulfilment of their goals and were mostly dropped because of lack of time.

The iterative development method was well suited to the project. The software was developed in time and the quality has satisfied its end-users, namely the partners in CRA and Universita di Calabria who developed the bioremediation code. It also worked well for the extension of the project, which was viewed as another cycle of the iterative process. Additionally, the repetitive walkthroughs of the CA Engine software, caused by the iterative development mode, benefited the quality of the final deliverables.

The software has also been validated using the Purify tool, which checks for safe memory manipulation. The claims for a bug-free environment are amplified by the rarity of bug reports tracked by the problem report mechanism, none of which was left open at the end of the project. Moreover, the software contains all the features the end-users have requested during the lifetime of the project using the problem reports mechanism mentioned in the previous section. The archive is available from the project's webpage (please note this is password protected):

```
http://www.epcc.ed.ac.uk/colombo/wp3/
```

The performance of the software was also highly satisfactory. Section 9 of this report is devoted to benchmarking results. The following are highlighted:

- The performance of the software scales up well with the number of processors;
- The bioremediation modelling problem calls for parallelism because it involves intensive calculations;
- The optimisations to the parallel code applied by EPCC improved the performance and scalability of the software.

1.4 Major Experiences and Lessons Learned

The most important issue in the lifetime of the project was probably the delay to produce the first software and documentation deliverables. This must be attributed to the aggressive schedule in the initial phases, which failed to predict the difficulty in the initial implemen-

tation and the volume of the work that had to be done. The initial schedule was also at fault, since the write-up of the first version of this document coincided with the Christmas of 1998. The reason why the schedule was so optimistic was the need for the other partners, the end-users of the deliverables, to use the software.

The iterative development method used came to rescue. EPCC, in association with the partners, preferred to delay the delivery rather than compromise the quality of the early deliverables. This was an important decision, given that software S1 and report D7 were the backbone of the final deliverables. The schedule did not deviate unacceptably though, since, in agreement with the partners, two internal delivery cycles were compacted into one, saving time for delivery preparation overheads without compromising the quality.

Another beneficial feature of the project was the excellent relationship between the partners. Especially for Workpackage 3 there was constant communication both for extra features and for problems with the software and the documentation. This was encouraged mainly by the problem report mechanism; however the frequent meetings made things easier and helped particularly to bridge the cultural gap between the partners.

Finally, we should not forget to stress the value that conferences added to the project. Apart from the obvious yet desirable effects of publicity and exposure a conference publication provides, the attendance helps put the project into context and import new ideas. The bold idea for computed fold boundaries, which never made it to be delivered because of lack of time, is a good example of high quality features that fertile conference attendances can suggest. Also, computational steering was put into context in an international "Problem Solving Environments" conference.

On the technical side of things there are many issues that Colombo highlighted and will be taken on board in the future. The availability of the software on various platforms was important but each platform should not be viewed in isolation. The use of XDR made CAMELot results portable as well as its code, providing a significant commercial advantage.

Another feature which was not seen initially as an important one, was the ability to run the software as a batch program without user interaction. This batch mode was central to the benchmarking of the software. It is also inevitable for the intended use of the software on a highly productive computing platform where submission queues are the only method available for the user to access resources.

Computational steering is also an important issue. The functionality is heavily used in the bioremediation simulation code. It must be stressed though that the availability of computational steering is an important and very desirable feature of many problem solving environments and is never straightforward to implement.

Perhaps one of the most important lessons learned has to do with the decomposition method use for Colombo. The initial regular decomposition assumed to be sufficient was anything but useful to the users who had to worry about the x-dimension divisibility with the number of processors (and folds). The issue was remedied in the last release of the software.

2. Introduction

CAMELot is an environment for the programming and seamlessly parallel execution of Cellular Automata. The system supports CARPET, a purpose-built language for CA programming. It offers a programming environment and a Graphical User Interface which enables the user to interact with the system while running a simulation and to view visualisations of the simulated data. It also includes a customisable facility to produce traces of the simulation in a specified format thus allowing to post-process the output of the run by means of an external tool. The system has been developed as part of the COLOMBO Project. It is a follow-up to the CAMEL software, implemented for the CABOTO project [Spezzano et al. 1995].

This document is the report on the implementation of CAMELot Release 1.3: Deliverable D9, Internal Deliverable DI3.6.2.

2.1 CAMELot Components

CAMELot consists of three major components:

- The CA Engine, incorporating a compiled CARPET CA model. This comprises one or more parallel processes called *macrocells* and uses an MPI-1-compliant message-passing library;
- The X/Motif-based graphical user interface (GUI), including the GUI/CA Engine communication library;
- The CARPET parser, which is integrated with the GUI.

An overview of the structure of CAMELot and the communication between its components during a running simulation is shown in Figure 1.

2.2 Software Components

The CAMELot implementation includes the following software components:

- `macrocell.c`
  The CA Engine module. Also contains code for the statistics output and random number generators.

- `libcmtguicomms.a`

  A library containing the GUI-related GUI-CA Engine communication functions. The source files are:

  - `guicomms.h`
  - `guicomms.c`

- `libcmtcommon.a`

  A library containing functions used in both the GUI and the CA Engine. The source files are:

  - `common.h`
  - `constants.h`
  - `list.c`
  - `plane.c`
  - `buffer.c`
  - `sock.c`

- `libcpt_parse.a`

  The library of CARPET parser-related functions. The source files are:

  - `parser.h`
  - `parser.c`
  - `cpt_parse.h`
  - `cpt_parse.c`
  - `yylex.l`
  - `yyparser.y`

- `camelot`

  The main CAMELot executable, including the GUI and parser. It is linked with the three libraries listed above, and is built from the following source files:

  - `camelot.h`
  - `camelot.c`
  - `camelot_stubs.c`
  - `camelot_viz.c`

```
  -  camelot_globals.c
```

2.3 Structure

The rest of this report discusses the components in turn. Section 3 contains the CAMELot User Manual. In section 4 we discuss the GUI implementation. In section 5 we deal with the CA Engine and in section 6 we give a brief description of the Parser. The communication protocol is discussed in section 7 and the Visualisation facility in section 8. Section 9 provides benchmarking results for the CA Engine and section **Errore. L'origine riferimento non è stata trovata.** lists the open issues of CAMELot. The release history is available from appendix I. The possibilities for MPI configuration can be found in appendix II.

2.4 Acknowledgements

The authors would like to thank Dr Mark Bull and Mr John Fisher for their contribution in this document. Dr Mark Bull has also contributed towards the testing and validation of the software.
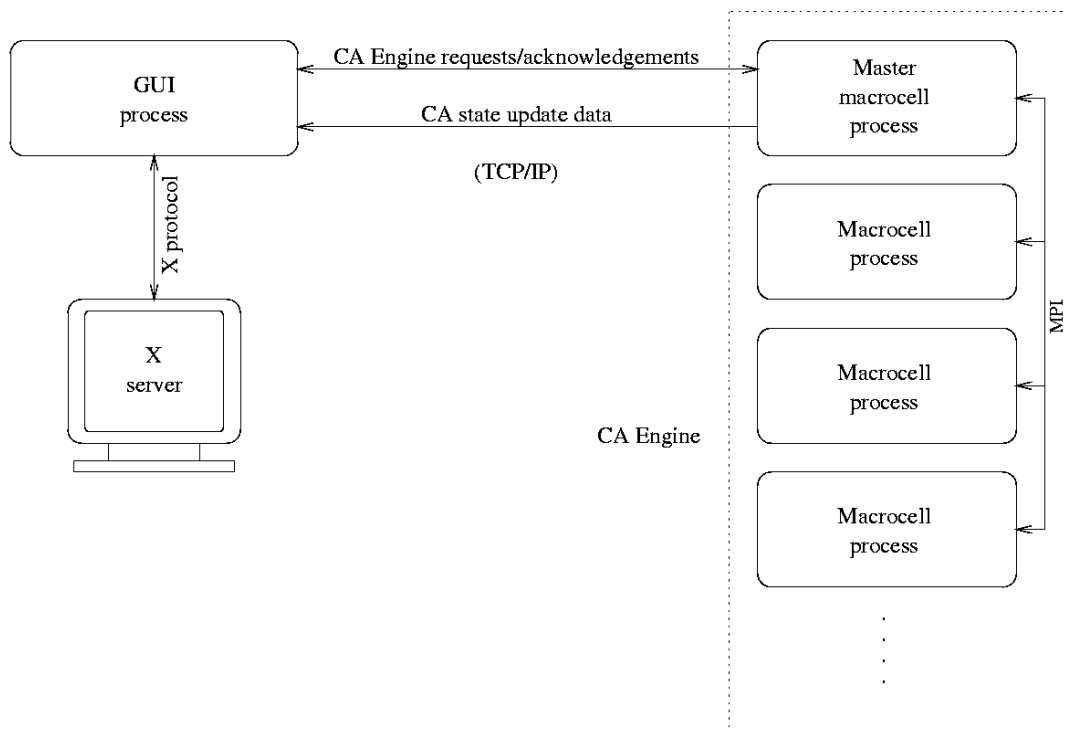
Figure 1: Overall structure of CAMELot

3.   User Manual

In this section we describe the functionality of CAMELot. We first provide an example of how CAMELot is run and use some of its basic features. We then give a detailed overview of CAMELot and finally we list and discuss the CARPET directives.

3.1 CAMELot Sample Session

3.1.1 Starting CAMELot

Assuming the current working directory is the top directory of the CAMELot binary distribution, CAMELot is invoked from a UNIX shell using the command:

> *platform*/`camelot` [*X options*] [*filename*]

Where *platform* is the platform identifier (the supported platforms are `sunos5`, `linux`, `irix6` and `tru64`); *filename* is a CARPET source file; and *X options* are the standard X application command line flags (`-display`,`-geometry`,`-iconic`,`-fn` etc). These command line arguments are optional. The CAMELot Development Window appears on the screen. It consists of three sections:

- A Menu Bar;
- An Editor subwindow with a scroll bar in each direction;
- A three-Button bar.

3.1.2 Editing a program

A user may write a program using the editor window. Alternatively, they may open a previously saved program file using the Open option of the File menu. After any modifications the file must be saved using the Save or Save As option of the File menu; if a filename has been provided, this is done automatically when pressing the Compile button.

Program editing is facilitated with the use of the Cut, Copy and Paste Options of the Edit menu. Shortcuts are available for all these functions.
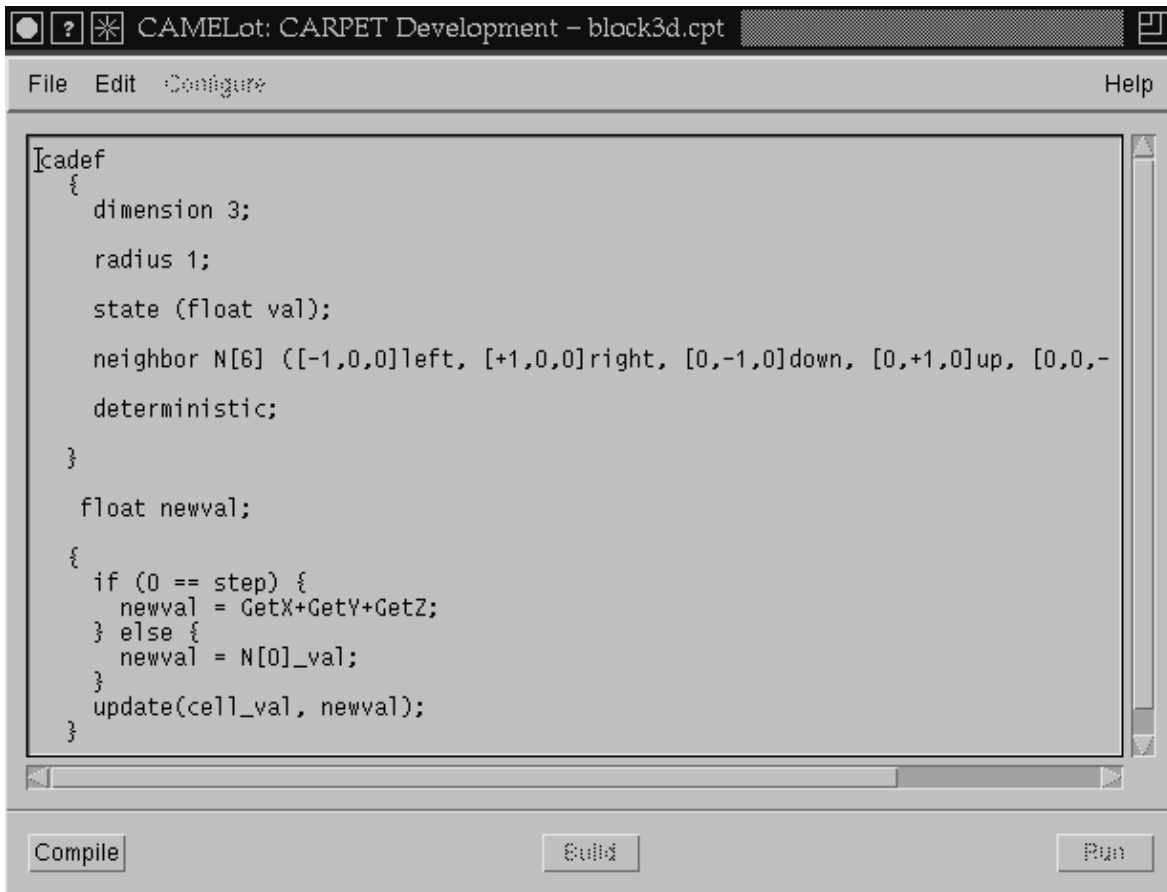
Figure 2: The Development Window

3.1.3 Program Compilation

When the program is ready, the user may compile it by clicking the Compile button. A successful compilation is followed by a pop-up window dismissible by clicking its Dismiss button. An erroneous compilation causes a beep and a pop-up window provides information about the error.
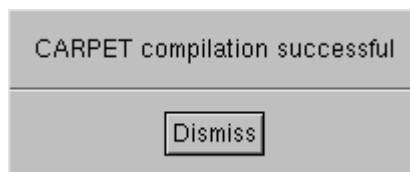
Figure 3: Successful Compilation Pop-Up Window

3.1.4 Building a Program

The Build operation generates a Unix executable file for CA execution. In order to build a file the user must first set the configuration parameters by using the Configure menu. These define:

- The Dimensions of the CA Engine;
- The number of Processes to handle the task;
- The number of Folds (see section 5.4 for more on folds) into which the task is divided.

The user can then build the executable by pressing the Build button. The output of the C compiler is shown to the user in a pop-up window.
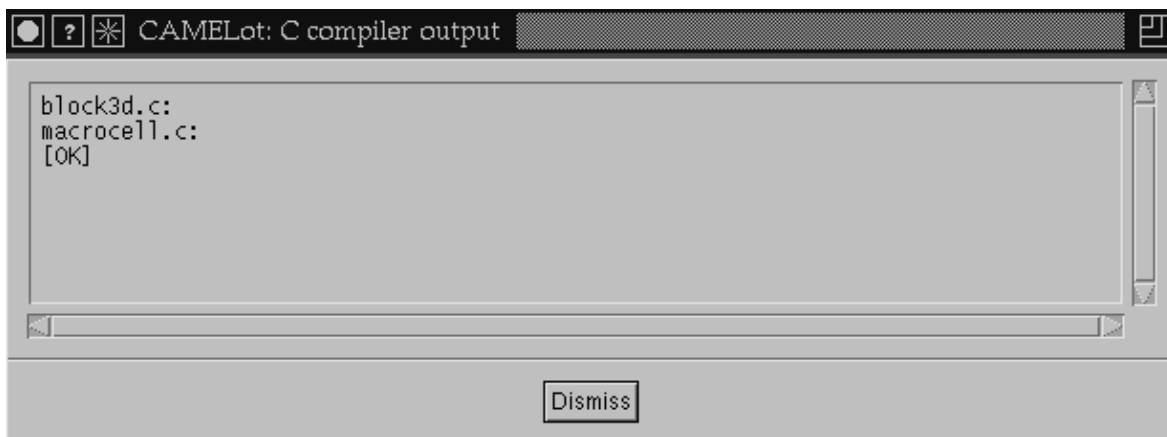
Figure 4: Successful Build Pop-Up Window

3.1.5 Running a Program

The Configure menu of the Development Window includes a menu by which the user can initialise the collection of statistics for the basic functions of the CA Engine. This should be enabled before clicking the Run button. After successful compilation and building the program, the user can invoke the executable by clicking the Run button. This pops up the Simulation Window which consists of three parts, a Menu bar, a Display part and a Button bar.
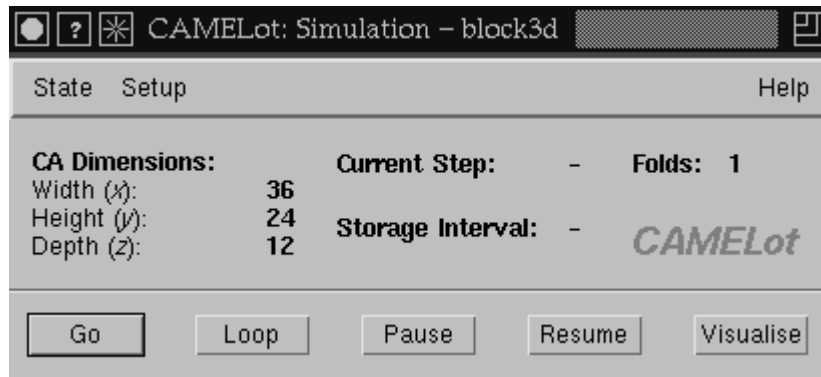
Figure 5: The Simulation Window

The State menu contains an Initialise and a Save Option. The user may initialise a substate or the whole state of a CA using an existing file, or save the current status of the CA. The Setup menu allows the definition of the number of CA evolutions to be run as well as other more advanced features, which are described later in this section. The display part of the window contains information about the configuration of the CA and updates the current step when the CA is running.

There are 5 buttons on this window. The Go and Loop buttons initialise the CA execution, the former for a number of steps defined from the Setup menu, the latter indefinitely (in fact for INT_MAX[1] steps). The Pause button temporarily suspends CA execution and allows visualisation window examination, state saving or editing etc. The user may continue the CA execution by clicking on the Resume button or restart the execution by clicking Go or Loop. The Visualise button allows the visualisation of a substate in various formats. The statistics for the functions of the system are output periodically during the run or after stopping the CA Engine execution, according to the user's request.

3.1.6 Exiting CAMELot

The user may close the Simulation Window and terminate the CA Engine execution by selecting the Close Option of the State menu. In order to exit CAMELot the user must select the Exit option in the File menu of the Development window.

---

[1] i.e 2^{31} -1 on 32-bit systems

3.2 CAMELot Functionality Overview

The CAMELot environment supports 3 different types of Windows. We will examine them in order of appearance when using the environment.

3.2.1 Development Window

The Development Window pops up when running CAMELot and, when it is closed, CAMELot exits. It consists of three parts, a Menu Bar, the CAMELot Editor and a 3-Button Bar.

3.2.1.1 Menu Bar

There are 4 pull-down menus.

Figure 6: The Development Window Menus. Note that the Configure Menu is greyed out since the screenshot was taken before compiling the file in the Editor.

- File;
- Edit;
- Configure;
- Help.

3.2.1.1.1 File

The File menu offers the following options:

- Open a file;
- Save a file;
- Save a file As;
- Load configuration;
- Save Configuration;
- Exit CAMELot.

The Open and Save As options pop up a window which allows the user to navigate through the filesystem and select the desired filename. For a file to be visible by Open, its name

must have the extension `.cpt`. The Save option is only available if a filename has been specified for a file being edited. The Exit button exits CAMELot; the Delete button usually available on X Window titlebars is disabled for this window.
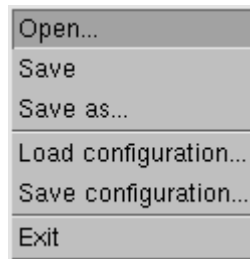
Figure 7: The File Menu

The characteristics of the program which are set under the Configure menu of the Development Window are automatically saved in a file named `progname.cnf`, `progname` being the full pathname of the CARPET file, every time the users saves the CARPET file. They are automatically retrieved when the CARPET file is Opened. In addition to this automatic facility, the Save and Load Configuration options allow the user to explicitly save and retrieve the configuration of the model[2].

3.2.1.1.2 Edit

The Edit menu contains the usual options. Keyboard short-cuts or "accelerators" (in brackets) are available for all options:

- Cut (Ctrl-X);
- Copy (Ctrl-C);
- Paste (Ctrl-V);
- Find (Ctrl-F);
- Find next (Ctrl-G);
- Replace (Ctrl-R).

Paste is only available after a Cut or Copy has been issued.

---

[2] Note that the `.cnf` file format was extended in CAMELot 1.2. `.cnf` files saved by CAMELot 1.1 are not compatible with CAMELot 1.2. When opening a CARPET file in CAMELot 1.2 which has a corresponding `.cnf` file saved by CAMELot 1.1, immediately check the Configuration menu settings and use "Save Configuration…" to overwrite the old `.cnf` file.

Figure 8: The Edit Menu

3.2.1.1.3 Configure

The Configure menu is made available after a successful compilation. It allows the user to modify the following parameters:

- The Dimensions of the CA (x-Length, y-Height, z-Width);
- The number of Processes to handle the task;
- The number of Folds to which the CA is divided in the Length axis.
- The C compiler pathname and flags;
- The MPI run command;
- The Timing output.

Figure 9: The Configure Menu

It is worth noting that:

- For best performance the Length of the CA must be an exact multiple of the product of the number of Processes with the number of Folds;
- A 1-D CA has only the x axis available and a 2-D CA has only the x and y axes available.

3.2.1.1.3.1 Controlling XDR Output

Starting from release 1.2 of CAMELot, XDR is used for the file I/O, this allows CAMELot data files to be portable between different machine architectures. The user can control the use of XDR through the use of the C compiler command line option of the Configure menu, shown in Figure 10.
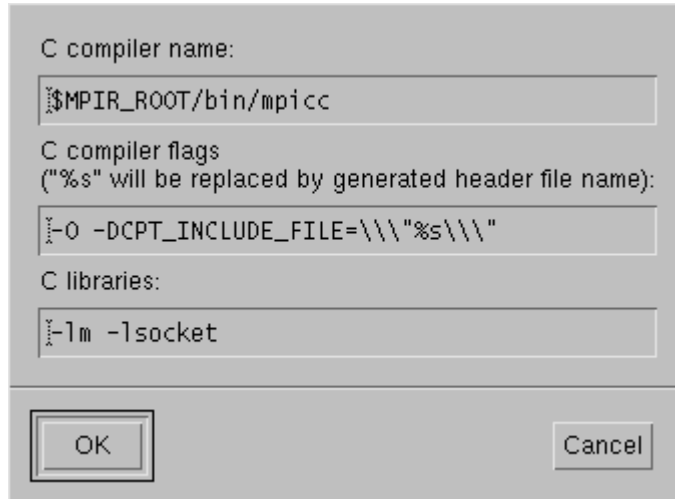
Figure 10: The C compiler command line option pop-up menu

- If the user wants to disable XDR for the operations related with reading from file, they should define -DNO_XDR_READ in the C compiler flags of the C compiler command line option pop-up menu;
- If the user wants to disable XDR for the operations related with writing to file, they should define -DNO_XDR_WRITE.

The user can use this facility to translate native binary project or substate files to XDR by applying the following:

- Load and compile the corresponding CARPET file;
- Specify the appropriate dimensions and define -DNO_XDR_READ in the C compiler flags of the C compiler command line option pop-up menu, then build;
- Click the Run button, load the configuration or substate files in question and then save them without running any iterations.

Please refer to the following sections for information about the steps mentioned in the above discussion.

3.2.1.1.3.2 Notes on Timing
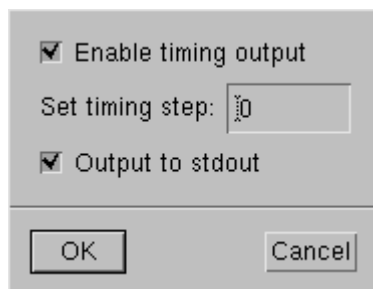
Figure 11: The Timer Configuration Menu

The CA Engine times its basic functions, namely the execution of the update statement, steering, visualisation, writing the system state on file and the total time spent excluding the time spent paused or stopped. The user can use the menu shown in Figure 11, which appears when the user clicks the Timing output option of the configure menu, to enable or disable the output of such results, set the period for printing them and direct the output to a file or the standard output of the terminal from which CAMELot was started. The format of the output is shown in Table 1.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Process: 0                                              Generations:

                        Calls            Time            Best            Worst
--------------------------------------------------------------------------------
Update Function :
Steering        :
Boundary Comm   :
Visualisation   :
Periodic Save   :

Sum             :
Total Execution Time:
```

Table 1: Output of Timing Statistics

Setting the timing step to zero results in the output being printed once after the CA Engine has been terminated (not at the end of the run). If the timing step is set to a value greater than zero, then the statistics are generated in the specified period. The time is accounted using double precision real numbers and is printed in floating-point representation in the standard C format (6 decimal digits), the measuring unit being seconds.

The Generations field contains the number of generations the output concerns. The field Calls counts the number of calls to each of the functions. Time is the total time taken for the calls in the Calls field. Best and Worst give the best and worst times for the function in question. The functions accounted are obvious. One remark is that after each iteration of the CA Engine, the read copy of the CA is updated by means of a `memcpy` call. Note that the time taken by this call is *not* accounted for by the Update Function timer. Sum gives the sum of the above times, whereas Total Execution Time counts the time total time taken, excluding the time spent paused or serving user requests. The Time field of Total can be less than the time shown in Sum, if the number of iterations is small, in which case the time taken for the initial boundary exchange is significant compared with the total execution time. This time is part of the Sum time but not part of Total.

3.2.1.1.3.3 Notes on Other Settings

The following settings can be made using the -D pre-processor flag in the "C compiler command line" menu option:
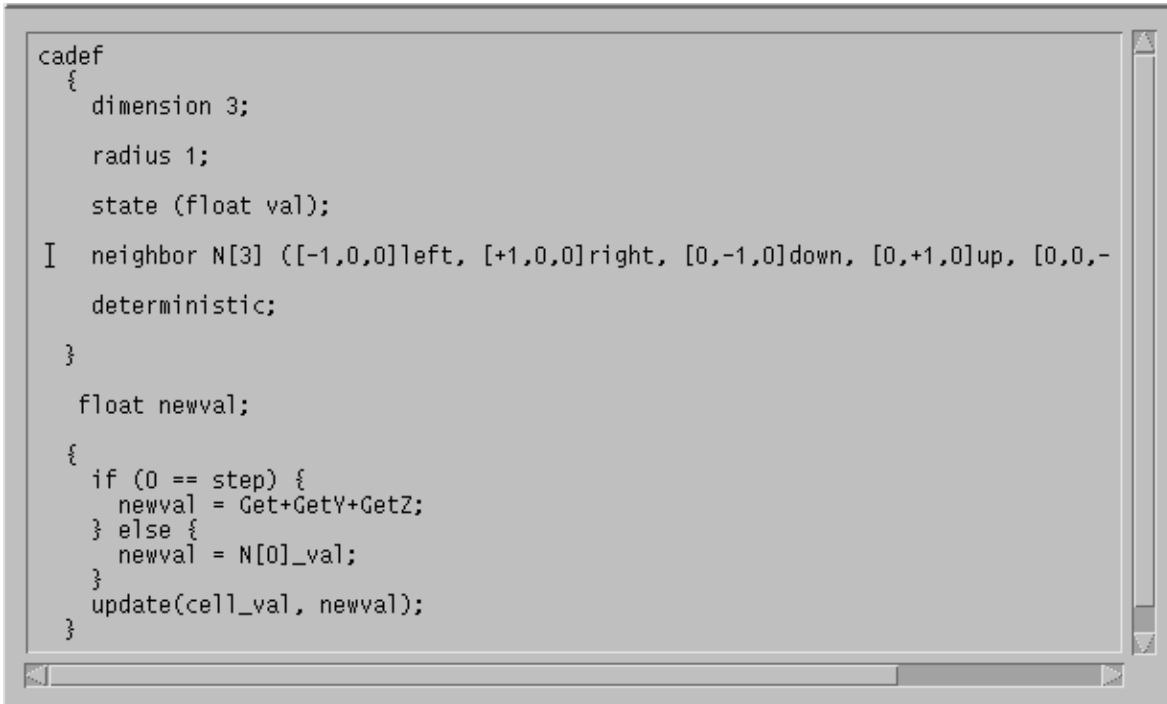
- `PROFILING`: This directive enables `gprof` profiling output. Its use is explained in [Kavoussanakis et al. 1999]. It should be noted that in order for this flag to be effective, the MPI libraries must be compiled with the `gprof` compiling enabled. The appropriate flag must be set in the "C compiler command line" as well.
- `DEBUG`: Provides assorted debugging messages
- `DEBUG_CALC_X_SIZES`: Ditto for function `calc_x_sizes()` (section 5.4.1).
- `DEBUG_CMT_READ`: Ditto for function `cmt_read()` (section 7.5.8.1).
- `DEBUG_CMT_WRITE`: Ditto for function `cmt_write()` (section 7.5.8.1).
- `DEBUG_CMT_BOUNDARY_SWAP`: Ditto for function `cmt_boundary_swap()` (section 5.5.2).
- `DEBUG_RUN`: Ditto for function `run()` (section 5.6.3).
- `DEBUG_TX_VIS_PACK`: Ditto for function `tx_vis_pack()` (section 8.6.2.1).
- `DEBUG_SERV_VIEW_STATE`: Ditto for function `serv_view_state()` (section 7.5.4).
- `DEBUG_SERV_SET_STATE`: Ditto for function `serv_set_state()` (section 7.5.4).
- `DEVELOP`: More assorted messages; it was used in the initial stages of developing the program.
- `EVEN_DECOMP`: Assumes that even decomposition of the model is possible. The effects of this are discussed extensively in sections 5.4 and 7.5.3 of this document.

- HOMOGENEOUS: A non-portable performance optimisation which is discussed in section 6.2.1.

3.2.1.1.4 Help

The details of the product authors are available from the About CAMELot option of the Help menu.

```
cadef
  {
    dimension 3;

    radius 1;

    state (float val);

    neighbor N[3] ([-1,0,0]left, [+1,0,0]right, [0,-1,0]down, [0,+1,0]up, [0,0,-

    deterministic;

  }

  float newval;

  {
    if (0 == step) {
      newval = Get+GetY+GetZ;
    } else {
      newval = N[0]_val;
    }
    update(cell_val, newval);
  }
```

Figure 12: The Editor of the Development Window. Note the two errors, the dimension of the `neighbor` vector and the undefined `Get` directive which will be detected in the Compile and Build processes respectively.

3.2.1.2 Editor Window

The user may Open a file and use the Editor to view and modify it (Figure 12). If the file exceeds the length (80 characters) or the height (24 characters) of the window, the user may use the respective scrollbars or the keyboard arrow keys.

3.2.1.3 Button Bar

The available buttons are:

- Compile;

- Build;
- Run.

Compile   Build   Run

Figure 13: The Button Bar of the Development Window. The only available button is compile since the screenshot was taken before compiling the file in the Editor and there was no configuration file available for this program.

Compile

This button compiles the current program in the Editor. This compilation checks for CARPET syntactic errors and generates the C source and header files for the specified CA model. The compiler handles both C (/* */) and C++ (//) style comments. A failed compilation is accompanied by a beep; a window is popped up containing the error messages and the cursor in the Editor is positioned at the first line reported to contain an error. If there is a beep but no error message is displayed then the automatic Save has failed.

ERROR: line 9: number of neighbor cells must be equal to neighbor vector size
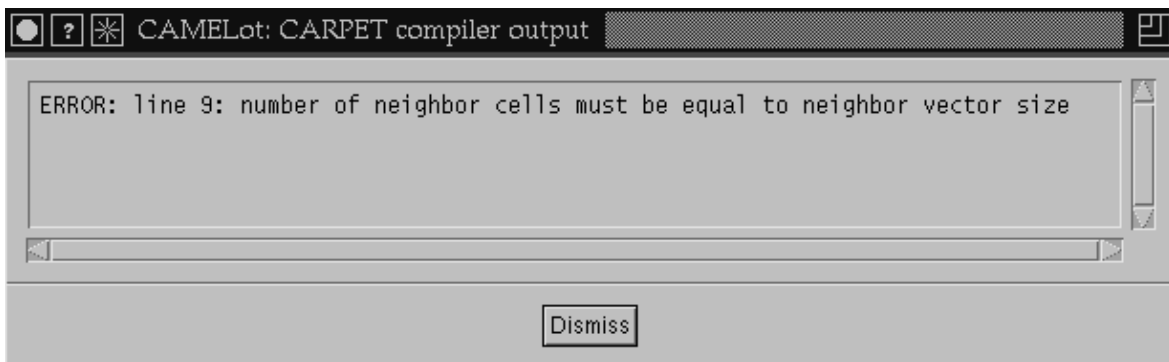
Figure 14: Error Message on the CARPET Compiler Output Window

If a compilation fails, the Build and Run buttons as well as the Configure menu are unavailable to the user.

N.B.: Clicking the Compile button in CAMELot 1.0 implicitly saved the CARPET file. This feature has been disabled in release 1.1 of the software to meet the users' request.

Build

This button compiles and links the CA Engine code with the generated C source and header files for the CA. It invokes the C compiler specified in the Configure menu and redirects its output to the pop-up window generated. Starting from release 1.1 of the software, the pop-up window contains an [OK] or [ERROR] line at the end of the message generated by the compiler, to provide feedback about the status of the finished compilation. This is helpful, because if the compilation is successful, the UNIX C compiler cc(1) usually generates no messages.

While the building of the program fails, the Run button is greyed out.

Run

This button spawns the CA Engine processes specified in the Configuration menu using the MPI run command as it appears in the respective option of the same menu. It also spawns the Simulation Window discussed next and makes the Build and Run buttons unavailable.
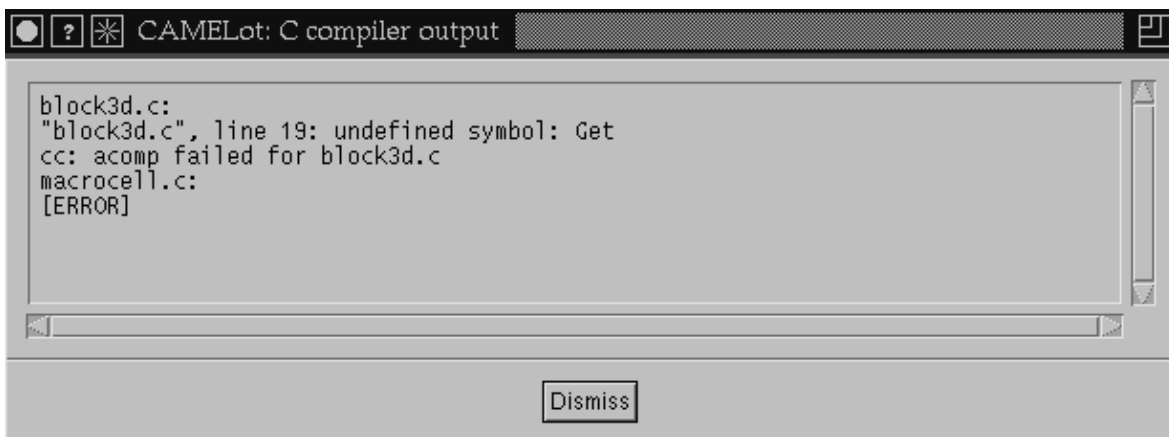
Figure 15: Error Message on the Build (C compiler) Output Window

3.2.2 Simulation Window

The Simulation Window comprises

- a Menu bar;
- a Display subwindow;
- a Button bar.

3.2.2.1 Menu Bar

The Menu Bar contains three menus:

- State;
- Setup;
- Help.

State   Setup   Help

Figure 16: The Menu Bar of the Simulation Window

3.2.2.1.1 State Menu

This allows the whole state or specific substates to be initialised or saved. The Close option closes the Simulation Window as well as all the Visualisation Windows and terminates the execution of the CA Engine.

A Substate can be saved in a binary file using the State-Save-Substate sequence of options. In order for the file to be subsequently detected as a substate file, it must be saved with the extension `.cmt`. Saving the Configuration involves saving status-specific data in a file with the extension `.cpj`, as well as all the substates in files with filenames constructed as follows: if the Configuration filename is `cfn.cpj` the substates are saved in filenames named `cfn000.cmt`, `cfn001.cmt`, etc.

Initialize
Save   ►
Close
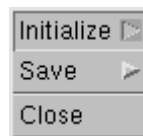
Figure 17: The State Menu

The data contained in a configuration file are:

- The number of Dimensions;
- The per-dimension Sizes;
- The current Generation of the CA Engine;
- The number of States;
- The number of Folds;

- The number of Global Parameters;
- The values of the Global Parameters.

Information stored in configuration or substate files can be loaded into the CA Engine using the State-Initialise options.

3.2.2.1.2 Setup Menu

The Setup Menu allows the following to be adjusted:

- Steps to run the Engine;
- Storage interval;
- Substate editing (one cell);
- Parameter editing;
- Active Fold setting;
- Manual setting of the per-substate minimum and maximum values for colour mapping.

Steps: Sets the number of CA Engine iterations to be run if the user presses the Go button.
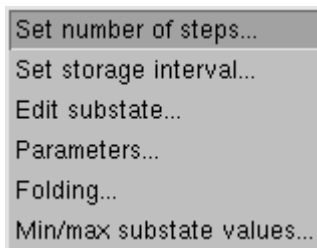
Figure 18: The Setup Menu

Storage Interval: Enables automatic CA Configuration saving with the set period. This involves saving the global parameters and other state variables in a binary file with the extension `.prj` and saving the substate data in files following the convention described earlier in the State Menu discussion. In addition to what stated there, the filename is prefixed with three characters denoting the sequence of the automatic save, starting with 000. If more than one thousand[3] consecutive automatic saves take place, the system overwrites the first without warning.

In addition to the above, an AVS field file is saved for each substate datatype, for each invocation of the macrocell executable (not each run, the file stops being updated when the

---

[3] In CAMELot 1.0 this limit was set to 100

Simulation Window is closed). These contain information to be used by the post-processing tool which is based on AVS/Express. The format of the field files, based in the AVS description [AVS 1993] is shown in Table 2.

To summarise the above, the following files are saved as a result of periodic configuration saves:

- In each save, a project file with the extension `.cpj`;
- In each save, a data file for each substate with the extension `.cmt`;
- In each simulation, a field file for each substate datatype with the extension `.fld`.

```
# AVS field file
# CAMELot generated
nstep = <number of expected⁴ saves>
ndim = <model dimension>
dim1 = <x-dimension>
dim2 = <y-dimension>
dim3 = <z-dimension>
nspace = 3
veclen = <number of associated substates>
data = <datatype of associated substates>
field = uniform
label = <names of associated substates⁵>

time value = 1
variable 1 file = <filename> filetype = binary
variable 2 file = <filename> filetype = binary


...
EOT⁶

time value = 2
...
```

Table 2: Format of CAMELot Generated AVS Field Files

For example, if the filename for the periodic configuration is `fname` and the system has three substates, two of which are of type `char` and one of type `float`, one periodic save will result to the following files being saved on disk:

    000fname.cpj

---

[4] This could differ from the number of actual saves if the user ends the run prematurely

[5] The format of the label list is described in the discussion of function `cmt_create_fld()` in section 7.5.8.1.

[6] Starting with release 1.3 of the software, the `EOT` separator appears between blocks of data refering to consecutive time steps

```
000fname000.cmt
000fname001.cmt
000fname002.cmt
fname_char.fld
fname_float.fld
```

It should be noted that after choosing the filename for the Project save, the CA Engine generates a warning if this choice will lead to files already on disk being overwritten. If the program is not running in Batch mode (see section 0 for details), this warning is also displayed in a pop-up window. The user can change their preference by repeating the operation described above; otherwise, the saves will occur.

Substate Editing: Allows the user to view and set the values of the substates of one cell manually. The possible substate names are made available through a menu.
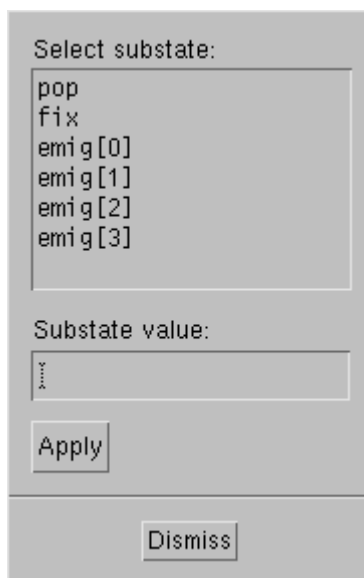
Figure 19: The Substate Editing Menu

Parameter Setting: Allows the adjustment of a global parameter. Parameters can be adjusted using the names they have in the program. The possible names are made available through a menu similar to the one shown in Figure 19.

Active Fold Setting: Allows the definition of the first and last active fold. This implies that the active regions can only be considered contiguous. Folds are numbered from 0 to NFolds-1; illegal values are disallowed. Alternatively, the automatic inactive region detection mechanism implemented in CAMELot allows non-contiguous active regions and

offers finer granularity. The mechanism is automatically activated if the CARPET program contains the statement `deterministic` and the user has not set the folds manually. Once deactivated, the automatic inactive region detection mechanism can be reactivated if the active folds are set to maximum range under the condition that the `deterministic` keyword exists in the CARPET program.
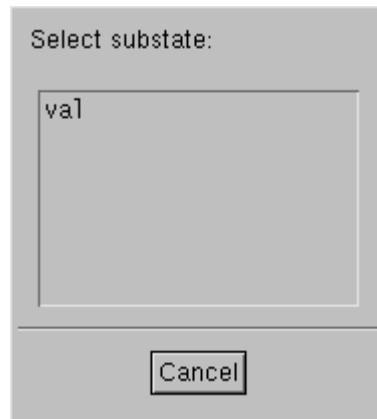
Figure 20: The Substate Selection Menu

Manual setting of the per-substate minimum and maximum values for colour mapping: This option allows the user to override the automatic per-substate minimum and maximum calculation executed as part of the colour mapping strategy. Specifying the minimum and maximum enables visualising parts of the data with greater detail. This does not affect the evolution of the model, although it speeds up the visualisation process. The system reverts to the automatic mechanism if the users clicks on the Auto button of the menu (Figure 21).
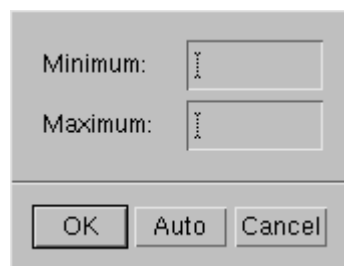
Figure 21: Manual Per-Substate Minimum and Maximum Value Setting Menu

3.2.2.2 Display Screen

Displays the current values of the following:

- The Dimensions;
- The Current Step of the CA Engine[7];
- The Periodic Storage Interval;
- The number of Folds. *N.B.:* "Folds: 1" indicates that no partitioning into multiple folds was done at compile time; i.e.. the CA is considered to consist of one single fold.

```
CA Dimensions:        Current Step:    –    Folds:  1
Width (x):       36
Height (y):      24   Storage Interval:  –   CAMELot
Depth (z):       12
```

Figure 22: The Display Part of the Simulation Window

3.2.2.3 Button Bar

The available buttons are:

- Go;
- Loop;
- Pause;
- Resume;
- Visualise.

```
Go        Loop        Pause        Resume        Visualise
```
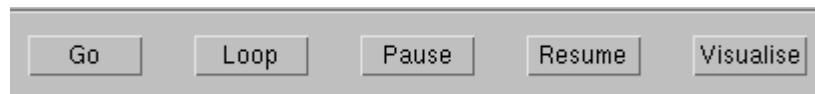
Figure 23: The Button Bar of the Simulation Window

Go: Starts the CA Engine until the generation counter reaches the number of iterations specified in the corresponding Setup menu option. It can be interrupted by State or Setup

---

[7] In the initial versions of the software this was available only if planes were visualised and only at the visualisation intervals. From release 1.1 this is available at all steps regardless of the existence of visualisation windows.

menu options as well as pressing any other buttons on the Button Bar *not* including Resume.

Loop: Same as Go except that it starts an infinite (`INT_MAX` iterations) CA evolution.

Pause: Temporarily suspends CA Engine execution. This can be restarted with any of three buttons.

- Go will restart the Engine until it reaches the specified number of iterations;
- Loop will restart the Engine for infinite iterations;
- Resume will continue the operation of the Engine from the step where it stopped. It will Loop if Loop was selected before Pause was pressed, or continue until the specified (possibly revised) finishing point is reached otherwise.

Visualise: Allows the initialisation of a Visualisation window. The user is prompted to set the visualisation period and select the substate to be visualised. In the case of a 3-D model the user has to select one of the three available visualisation formats discussed in the Visualisation Window Section.

N.B.: As of the release 1.1 of the software, Go and Loop no longer zero the iteration counter.

3.2.3 Visualisation Window

The Visualisation Window comprises two basic parts:

- the Visualisation Space;
- the Button Bar.

Important information is also displayed in the title bar of the X window, namely the CARPET program filename, the Visualisation Step, the name of the visualised Substate, and the entity Coordinates.

3.2.3.1 Visualisation Space

This occupies an area of 640x640 pixels (not including the colour palette bar area). The visualised entity is scaled so as to fit in the window. If the size (in cells) of the visualised entity is too big to represent each cell by at least one pixel, then the cells of the entity are

sampled at regular spatial intervals. These sampled cells are drawn as single pixels. No averaging over the interval is performed.

The user can resize the visualisation window. This is achieved by means of the corresponding facility of the user's Window Manager. If the user decreases the visualisation space, scroll bars appear at the right-hand and bottom sides of the window. The default size of the window is the maximum; increasing the size further does not make the visualisation larger. However, it is meaningful for a decreased window to be enlarged at will, until it reaches its maximum size. This happens because resizing the window does not cause the visualised entity to be zoomed in or out, it only moves the borders of the window.

The colour palette currently in use is shown as a horizontal bar at the bottom of the Visualisation Space of the window. The minimum and maximum values for the visualised sub-state are shown above this bar.

The possible types of visualisation depend on the number of dimensions of the model:

1-D Models: The visualisation is drawn in horizontal lines from left to right. The vertical dimension of the window corresponds to time. The user can therefore see how the model changes with time. When the vertical dimension of the screen is exhausted, the visualisation restarts from the first line overwriting the first visualisation.

2-D Models: They are represented in an orthogonal manner, *x* running horizontally and *y* running vertically, the origin being the bottom left corner of the window.
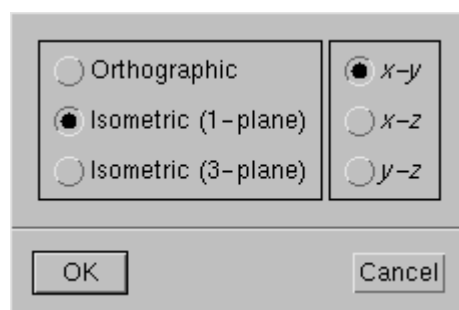
Figure 24: The Possible Types of Visualisation of a 3-D Model

3-D Models: x-y, x-z or *y-z* planes of a 3-D model can be displayed either as orthographic (as above) or isometric projections. The coordinate of the plane (i.e.. *z* value for an *x-y* plane, *y* value for a *x-z* plane etc) is specified by the user via a dialog box with scale widgets.

In the orthographic case, x-y planes are displayed as above, x-z planes are displayed with a horizontal x-axis and vertical z-axis, and y-z planes are displayed with a vertical y-axis and horizontal z-axis. In the isometric case, the y-axis is oriented vertically, the x-axis is oriented upper-left to lower-right and the z-axis lower-left to upper-right.
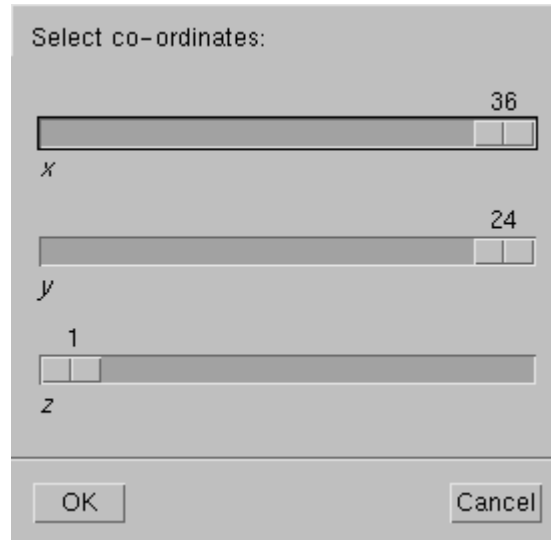
Figure 25: Plane Coordinate Dialog Box

A 3-plane isometric view is available for 3-D models. In this case, an x-y, x-z and a y-z plane are selected by the user. If the coordinates of these planes are denoted z_1, y_1, and x_1 respectively and the size of the CA in the z dimension is z_{max} then the x-y plane from $x=0$ to $x=x_1$ and $y=0$ to $y=y_1$, the x-z plane from $x=0$ to $x=x_1$ and $z=z_1$ to $z=z_{max}$ and the y-z plane from $y=0$ to $y=y_1$ and $z=z_1$ to $z=z_{max}$ are displayed as three faces of a cuboid with the axes oriented as for the 1-plane isometric case. The origin is thus the lower leftmost visible vertex, i.e.. a "left-handed" coordinate system is used.

From the above it can be deduced that in order to visualise a substate for the entire 3-D model, the user has to select the x and y coordinates to be equal to the maximum value for the x and y dimensions respectively and $z=1$ ($x_1=x_{max}$, $y_1=y_{max}$, $z_1=1$), as shown in Figure 25.

3.2.3.2 Button Bar

This contains two buttons:

- Colours;
- Close.

Colours: Allows the user to set the 256-colour palette to match their preference. The default is coloured from blue (lowest value) to red (highest value), the intermediate values mapped to cyan, green and yellow in ascending order. A monochrome (greyscale) palette ranging from black to white is also available. The files specifying the palette are stored with the extension `.pal` and contain 256 lines with 3 space-separated unsigned 16-bit hexadecimal values for Red, Green and Blue respectively, in each line. The colour in the first line is used for background. The palette selected is shown in the colour palette bar.

Close: Closes the Visualisation Window; this does not affect the CA Engine execution. The user may also close the Window from the Delete button of the Window Manager.

A screenshot of the Visualisation Window for the model in Figure 12 is shown in Figure 26.
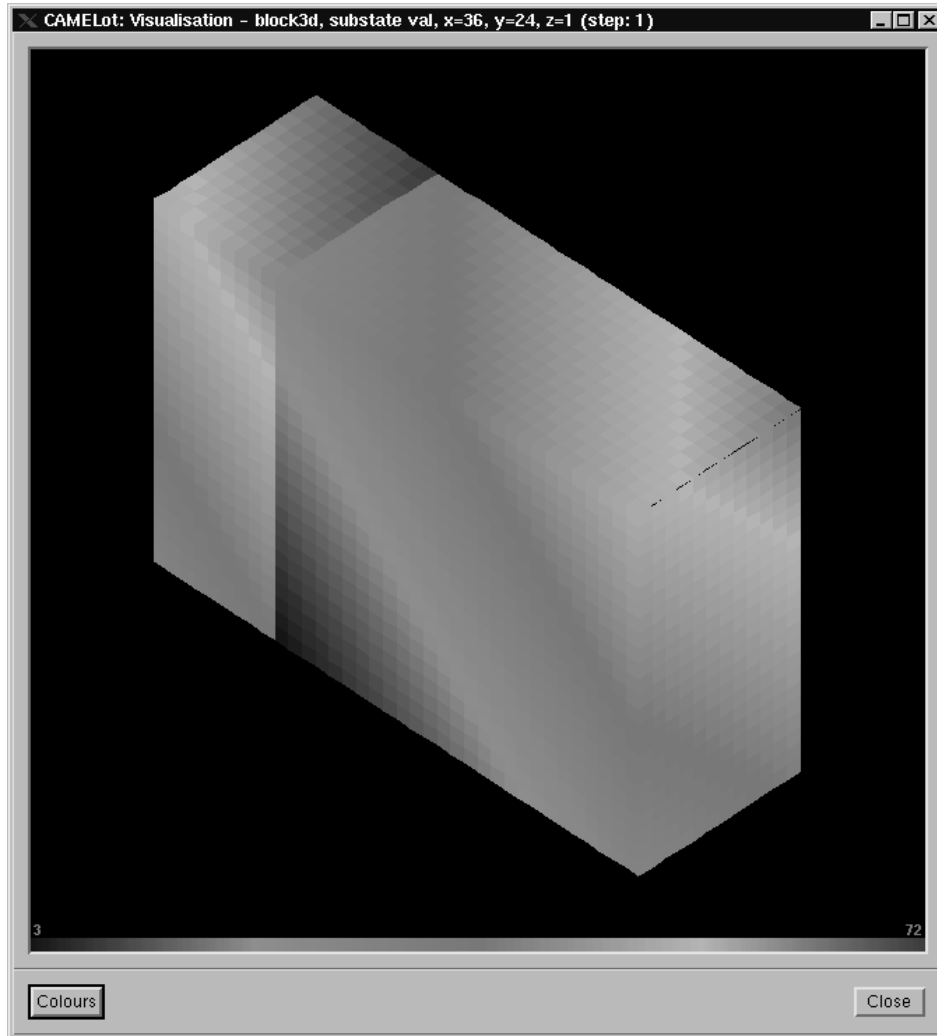
**Figure 26: The Visualisation Window. This is the output of the model in Figure 12
after 10 iterations of the CA Engine.**

3.2.4 Off-line CA Engine Execution

The CA Engine can be invoked outside the CAMELot environment. Limited functionality
is supported. In the following discussion we assume that the user has built the CA executable either using the CAMELot environment or using the Makefile.batch makefile available
with the distribution.

The command-line arguments available to the user are as follows:

-l[8]<no_of_state> <filename>

---

[8] This is the letter "*el*". No space exists between the "l" and the state index.

Initialise substate `<no_of_substate>` from `<filename>`. This suggests that the user knows the substate index allocation done transparently in the CARPET parser. These indices can be deduced from the `state` CARPET statement, as they are parsed sequentially, i.e. the first substate given is indexed 0, the second 1 etc.

`-0`

This initialises all substates to 0.

`-n<num_gens>`

Set the number of generations to be run to `<num_gens>`.

`-s<save_step> <filename>`

Enable periodic project save to files with basename `<filename>` (according to the conventions for saving a project) with period `<save_step>`. See section 3.2.2.1 for more details.

`-t<time_step> <filename>`

Enable periodic timing statistics output to file `<filename>` with period `<time_step>`. If `<time_step>` equals zero, then the results are output at the end of the simulation. If `<filename>` is set to `-`, then the results are written to the standard output of the terminal window where CAMELot was started. See section 3.2.1.1 for more details.

`<filename>`
Initialise the project from `<filename>`. This has no parameter to identify it and it must be the last argument. If `-l` or `-0` have been specified it is ignored.

3.3 The CARPET Programming Language

CARPET is a programming language for the definition of cellular automata-based models and their transition functions, designed as an extension to ANSI C. A CARPET program consists of the following sections: a global declaration section, known as the *cadef* (CA DEFinition) section; a transition function; and an optional *steering* function.

The general layout of a CARPET program is as follows:

```
cadef
{
      declarations
}
```

[*transition function local variable declarations and subroutine prototypes*]

```
{
      transition function code
}
```

[*transition function subroutines*]

```
[
steering
{
      steering function code
}
]
```

where items in […] brackets are optional. Note that the steering function must be located *after* the transition function and any subroutine functions called by the transition function.

The extensions to C defined in the CARPET language are described below.

3.3.1 The transition function

The transition function (and its subroutine functions, if any) may contain the following CARPET statements, in addition to C code:

- cell_*substate*
- DimX, DimY, DimZ
- GetX, GetY, GetZ
- NFolds
- NProcs
- random()
- randomise()

- `srandom()`
- `step`
- `update()`
- parameter references

3.3.2 cadef

Syntax

```
cadef
{
        declaration;
        declaration;
               ...
        declaration;
}
```

Remarks

This is the declaration section of the program; it must precede any statement except for C pre-processor ones. `declaration` can be any of the following statements:

- `deterministic`
- `dimension`
- `neighbour`
- `parameter`
- `radius`
- `region`
- `state`
- `threshold`

Example

```
cadef
{
    dimension 3;
    radius 1;
    region Inside (start+1:end-2, :,:);
    state (float val; int val2);
    neighbour N[6] ([-1,0,0]left,[1,0,0]right,
                    [0,-1,0]down, [0,1,0]up,
                    [0,0,-1]in, [0,0,1]out);
    parameter (pi 3.14159);
    deterministic;
    threshold (cell_val == 3);
}
```

3.3.3 cell_<substate>

Syntax

```
cell_<substate>
```

Remarks

A user may refer to a specific substate of a cell by using the string "cell_" followed by the name of the substate.

N.B.: A user may modify the value of a substate using the update function (section 3.3.25).

Example

```
cadef
{
    state (float temp);
}


float val;


val = cell_val+3;
```

3.3.4 cpt_abort

Syntax

```
cpt_abort()
```

Remarks

Calling this function causes the program to stop. It is only available inside the `steering` block of the program.

Example

An example is available in section 3.3.22.

3.3.5 cpt_save

Syntax

```
cpt_save (char *fname)
```

Remarks

This function saves all the CA Engine data to project and substate files. It does not save the AVS/Express related files. It is only available for steering. It uses the `fname` argument as a root for the generated files, as described in section 3.2.2.1.2 (omitting the AVS/Express related discussion).

Example

An example is available in section 3.3.22.

3.3.6 cpt_set_param

Syntax

```
cpt_set_param (float *par, float npar)
```

Remarks

This function alters the value of the global parameter pointed by `par` to that of `npar`. It is only available inside the `steering` statement. See section 3.3.14 for the definition of global parameters.

Example

An example is available in section 3.3.22.

3.3.7 *deterministic (alias determin)*

Syntax

```
deterministic;
```

Remarks

This statement signifies the commitment of the programmer that the cell update function is deterministic [Telford et al. 1998]. A deterministic program is one where the state of a cell is guaranteeed to be unchanged if the state of its local neighbourhood is unchanged. This is one of the necessary conditions for automatic inactive region detection (the other being that the user has not set active folds manually).

N.B.: A program whose update rule depends on `step` or random functions is non-deterministic (except if this only happens in step 0). Starting with release 1.2 of the software, detection of the `deterministic` keyword and the keyword `step` or any random function in a program is flagged by the parser as a warning (non-fatal).

Example

The example below gives an example where the incorrect use of `deterministic` leads to erroneous program execution.

```
cadef {
    ...
    state (float val);
    deterministic;
}

float newval;
{
    if (0 == step) {                // OK
        newval = GetX+GetY+GetZ;
    } else if (step < 20) {        // not deterministic!
        newval = 0.51*cell_val;
    }
    ...
    update(cell_val, newval);
}
```

3.3.8 dimension

Syntax

```
dimension <n>;
```

Remarks

Defines the number of dimensions of the CA Engine. It ranges from 1-3.

3.3.9 DimX, DimY, DimZ

Syntax

```
DimX, DimY, DimZ
```

Remarks

These values are the CA Engine dimension of the *x*, *y* and *z* axis respectively. Note that for *x*, which is split according to the number of processes, this is the size of the whole model.

Example

```
cadef {
    ...
    state (int st);
    ...
}


{
    DimX = 5;              // This is illegal!

    if (DimX == cell_st) {
        update(cell_st, DimY);
    }
    ...
}
```

3.3.10 *GetX, GetY, GetZ*

Syntax

```
GetX, GetY, GetZ
```

Remarks

These values are the global *x*, *y* and *z* coordinates of the cell respectively.

Example

```
cadef
{
    ...
    state (float dist);
    ...
}


float val;
{
    if (0 == step) {
        val = GetX+GetY+GetZ-3;
        update(cell_dist, val);
    }
    ...
}
```

3.3.11 *neighbour (alias neighbor)*

Syntax

```
neighbour <Nname>[<n>](<[x,y,z] alias>, ...,
                      <[x,y,z] alias>);
```

Remarks

This statement defines a logical neighbourhood. The x,y,z values must remain within the [-radius, radius] interval defined in the cadef statement. The alias for each neighbour is not compulsory; a cell can refer to its neighbour using the Nname[i] notation.

Example

```
cadef {
     dimension 2;
     radius 1;
     state (float dist);
     neighbour Neumann[4]([0,-1] North, [0,1], [-1,0],
                             [1,0]);
}


float v1, v2;
{
     v1 = North_dist;
     v2 = Neumann_dist[0];    // Should be the same!
     ...
}
```

3.3.12 NFolds

Syntax

```
NFolds
```

Remarks

Returns the number of folds.

Example

```
cadef
{
     ...
}


NFolds = 3;          // illegal!
if (1 == NFolds) {
...
```

3.3.13 NProcs

Syntax

```
NProcs
```

Remarks

Returns the number of processes to which the CA is split in the *x* axis.

Example

```
cadef
{
    ...
}


Nprocs = 2;            // illegal!
int strip_length;


strip_length = DimX/(NProcs * NFolds);
...
```

3.3.14 parameter

Syntax

```
parameter (param_def_list)
```

where `param_def_list` is a comma-separated list of parameter definitions, where each parameter definition has one of the following forms:

- `param_name`
- `param_name value`
  where `value` is a `float`, in any C `float` syntax.
- `param_array[dim]`
- `param_array[dim] {array_list}`
  where `array_list` is a comma-separated list of `floats`, in any C `float` syntax, and `dim` is an `integer` index greater than 1.

Remarks

Declares and defines global CA parameters. Their values can only be changed during the run from the GUI, or by means of the `cpt_set_param()` primitive (section 3.3.6), since they are global to all the cells. They are of type `float`. Parameters are accessed in a CARPET program directly through their symbolic name. The maximum number of parameters (counting each element of parameter arrays) is 500, as set by the `MaxNumParam` variable in the file `parser.h` of the parser. The same file contains the definition of the maximum length of a parameter name (30

characters, including the array indices) `MaxLenParam`. The array list may have fewer than `dim` elements, in which case the additional values default to zero (non-initialised parameters default to 0 in any case).

Example

```
cadef {
      ...
      parameter (mypar 2.0, par_array[3] {1.0, 4.0});
      ...
}
```

3.3.15 radius

Syntax

```
radius n;
```

Remarks

Defines the radius of the neighbourhood of the cells.

N.B.: `radius` is limited to

- 60, if `dimension` is 1;
- 2, if `dimension` is 2;
- 1, if `dimension` is 3.

3.3.16 random

Syntax

```
random (n);
```

Remarks

Returns a pseudo random integer between 0 and `n`, n being a positive integer.

N.B.: `random()` returns the same sequence of numbers every time it is called. To avoid this, the user may use the `randomise()` function. The use of this function could make a program non-deterministic (see section 3.3.7).

3.3.17 randomise (alias randomize)

Syntax

```
randomise();
```

Remarks

Creates a new seed for the random number generator.

3.3.18 region

Syntax

```
region <region-name> (<min_x>:<max_x>,<min_y>:<max_y>,
                      <min_z>:<max_z>);
```

Remarks

The user specifies a region as part of the `cadef` block of the program, using a declaration of the above form. This is used to allow global reduction operations within the steering block of the CARPET program. There is no limit to the number of regions that can be specified by the user. If the lower or upper bound of a co-efficient of the region is not defined, the specification defaults to the corresponding minimum or maximum for the respective dimension. The bounds of the region range from 1 to the size of the corresponding dimension. The keywords `start` and `end` are defined to be the minimum and maximum of the dimension in which they are found, thus allowing flexible region specification.

Because the dimensions of the model are specified at build time while the regions are declared at compile time, full error checking is not possible. Nonetheless, the following conditions are flagged as errors:

- Specifying minimum and maximum values for dimensions not used by the model;
- Specifying a negative integer as a range boundary;
- Specifying a maximum value less than a minimum value (this check is possible if the region boundaries are explicitly defined).

Example

```
cadef {
    dimension 3;
    region myregion (start+2:end-2,:,3:);
    ...
}
```

3.3.19 region_<op>

Syntax

```
region_<op> (<region-name>, <state>);
```

Remarks

The `region_<op>()` function is available inside the steering function. It returns a value of the same type as its `state` argument. It applies the reduction operation `op` to state `state` all the cells in region `region-name`. The supported operations are as follows:

- `max`
- `min`
- `sum`
- `prod`
- `land` (logical and)
- `band` (binary and)
- `lor` (logical or)
- `bor` (binary or)
- `lxor` (logical exclusive or)
- `bxor` (binary exclusive or)

The user can supply a global reduction operation inside their CARPET program to cater for operations other than the ones above. The prototype of a global reduction function corresponding to the `region_<op>()` function must comply with the following:

```
<datatype> cpt_<datatype>_<op> (int min_x, int max_x,
int min_y, int max_y, int min_z, int max_z, int stateid,
CptCell *cp)
```

N.B.: The automatically generated functions assume that the co-efficients of the model are in the [0,DIMw-1] range, w $\in \{X,Y,Z\}$, in other words, they range from 0 to the maximum dimension of the model minus 1.

Note: the importance of the neutral element

When developing a global reduction function the user should take into account that a region can be defined so that the data in it are outside the domain of one or more

processes. The functions automatically generated provide an algorithm which prevents erroneous calculation. However, because the operations are global, all the processes contribute to the global reduction of the result. In order to avoid incorrect global reduction of the data, the user can set the initial value of the variable containing the per-process result of the function to be equal to the neutral element for the corresponding operation.

Example

Examples of such functions are the ones automatically generated by the parser.

3.3.20 srandom

Syntax

```
srandom (n);
```

Remarks

Same as `randomise()`, only that the programmer may choose the seed argument through n.

3.3.21 state

Syntax

```
state(type substateA1, substateA2, …, substateAn,
      type substateB1, substateB2, …, substateBn, …);
```

Remarks

The state of a cell consists of various typed substates. The allowed types are:

- (unsigned) char;
- (unsigned) short;
- (unsigned) int;
- float;
- double;
- arrays of the above.

3.3.22 *steering*

Syntax

```
steering {
    statement;
    ...
    statement;
}
```

Remarks

The steering function is an optional feature of a CARPET program by which the user can affect the flow of the program as a result of global reductions on regions of the model (see section 3.3.18 for the definition of regions in a CARPET program).

The steering function is defined in a separate section of the CARPET program, similarly to the update function. *The main difference is that the update function is applied separately in each cell, whereas the steering function is global for the model.* Any code inside the `steering` statement is copied verbatim to the generated file, with the exception of the `region_<op>()` statements which are translated to a global reduction function, as shown in section 3.3.19. The user can modify the flow of the program inside the steering section in either of the following two ways:

- call the function `cpt_set_param (float *old_p, float new_p)`, which sets the global parameter pointed by `old_p` to the value of `new_p`;
- call the function `cpt_abort()`, which terminates the execution of the program without exiting the CA Engine.

Inside the steering code, the user has access *only* to the following CARPET defined variables:

- `DimX`, `DimY`, `DimZ`;
- `step`;
- global parameter values.

Example

```
cadef {
  dimension 3;
  region Inside (start+2:end-2,:,:);
  ...
  state (float val);
  parameter (pi 3.141);
}
...
steering {
  float min = region_min (Inside, val);

  if (min < 4.0) {
    cpt_set_param (&pi, 3.14159);
  } else if (min > 100.0) {
    cpt_save ("aborted");
    cpt_abort ();
  }
}
```

3.3.23 step

Syntax

```
step
```

Remarks

This denotes the current CA Engine iteration. The initial value is 0. Allows time-dependent update function development.

3.3.24 threshold

Syntax

```
threshold (expression);
```

Remarks

Defines a C expression, which, if satisfied, is equivalent to the cell being idle for the past CA Engine evolution. It is used in conjunction with `deterministic` for the inactive region detection.

Example

```
cadef
{
    ...
    state (  float val;  int val2; );
    threshold (cell_val == 3);
    ...
}
```

3.3.25 update

Syntax

```
update (cell_substate, value);
```

Remarks

This is the only way to set the value of a cell substate by means of the program. This is done in order to ensure that the state of all cells is set in lock step in the next generation *after* the update has been issued, thus preventing race conditions.

4. GUI Implementation

4.1 Overview

The CAMELot GUI is a Motif application written using Imperial Software Technology's X-Designer 4.6 and 5.0 GUI builder tool, which is a tool for designing GUIs graphically. This tool generates (in this case) C code which implements a GUI using the Motif library.

This approach was taken to allow rapid prototyping and development. Since X-Designer also has facilities for generating C++ (using Motif or Microsoft MFC libraries) or Java (using the AWT library), this may also ease any future porting of CAMELot to other platforms.

This tool has been used to generate code to implement the visual elements of the GUI (text-editing widget, menus, buttons, etc). The rest of the GUI functionality (CARPET file reading and writing, CARPET compilation and building, communication with the CA Engine, visualisation) was coded by hand and integrated with the X-Designer-generated code by means of X callback interfaces. The CARPET parser and the CA Engine communication module were written as separate libraries linked with the GUI. These are described elsewhere in this document.

4.2 Communication with the CA Engine

Communication with the CA Engine is via two Internet-domain sockets, `prot_sockfd` for CA Engine requests and acknowledgements and `vis_sockfd` for receiving visualisation data and the current generation number in each step. These are opened when the CA Engine is spawned and closed when it is terminated.

The incoming visualisation data socket is multiplexed into the X event loop using the `XtAppAddInput()` X Toolkit function.

4.3 Visualisation Windows

The Visualisation window's main graphics area is implemented in hand-coded Xlib pixmap code (for efficiency) within an XmDrawingArea widget.

CAMELot uses the default X Visual for the X Screen it is displaying on. CAMELot supports the following types of Visuals: PseudoColor, DirectColor or TrueColor with colour depth of at least 8 bits. A separate colormap for each Visualisation window is set using `XSetWindowColormap()`; this allows different windows to use different colour palettes if desired. Hence, a display of a depth greater than 8 bits is recommended to avoid colormap switching when changing window focus.

The default colormap is a red/yellow/green/blue spectrum plus black for the background.

4.4 Source code files

The following source files comprise the GUI:

- `camelot.c` (partly X-Designer generated)
  This contains the `main()` function. This file has been hand-edited to set a fallback X resource in order to override the CDE Motif default `*FontList` resource and to open a CARPET source file on startup if one is specified on the command line.

- `camelot.h`
  This contains global `cpp` definitions and declarations. Some of the optional compile-time `cpp` flags defined in this file are:

  `DEBUG_CROAKS`: Enables assorted debug trace statements to `stderr`

  `DEV_CONFIG`: Development configuration (e.g. no "Save CARPET file?" dialog box on exit)

  `LOCALHOST_SOCKET`: Hardwire front-end hostname to "`localhost`" (this is the hostname passed to the CA Engine in order for it to initiate the connection to the GUI). This is a useful optimisation if the GUI and CA Engine run on the same IP host.

- `camelot_ext.h` (X-Designer generated)
  Widget declarations.

- `camelot_globals.c`
  Global variable definitions.

- `camelot_gui.c` (X-Designer generated)
  Widget creation/deletion code.

- `camelot_stubs.c` (partly X-Designer generated)
  X-Designer-generated GUI callbacks plus all non-X-Designer-generated code, except for Visualisation window code.

- `camelot_viz.c`
  Visualisation window code.

- `Makefile.{sunos5,irix6,linux,tru64}` (partly X-Designer generated)
  Makefile for the CAMELot GUI for various platforms (SunOS 5.6, IRIX 6.2, Red Hat Linux 5.2 and Tru64 UNIX 4.0F respectively).

4.5 Libraries

The GUI is linked with the following libraries, which form part of the whole CAMELot system. These are described elsewhere in the document:

- `libcpt_parse`   The CARPET parser
- `libcmtguicomms`   CA Engine communication interface
- `libcmtcommon`   Code common to GUI and CA Engine

4.6 X Shell Widgets

The following X shell widgets are defined in the X-Designer design:

- `dev_shell`
  Development window.

- `sim_shell`
  Simulation window.

- `viz_shell`

  Visualisation window. A shell widget for each Visualisation window opened by the user is created by calling the X-Designer-generated function `create_viz_shell()` in `viz_open()`.

- `devdims_shell`

  Development window Configure menu "Automata dimensions..." dialog box.

- `devcc_shell`

  Development window Configure menu "C compiler command line..." dialog box.

- `devmpi_shell`

  Development window Configure menu "MPI run command..." dialog box.

- `devreplace_shell`

  Development window Edit menu "Replace…" dialog box.

- `devcomp_shell`

  CARPET parser (compiler) output window.

- `devbuild_shell`

  C compiler output window.

- `about_shell`

  "About CAMELot..." message box.

- `filesel_shell`

  File selector dialog box.

- `substate_shell`

  CA substate selector dialog box.

- `cell_shell`

  CA cell selector dialog box (x, y, z coordinate scales).

- `fold_shell`

  CA active fold selector dialog box (Simulation window Setup menu "Folding..." dialog box).

- `disptype_shell`

  Display type (orthographic, isometric 1-plane or isometric 3-plane) selector dialog box.

- `minmax_shell`

  Simulation window "Min/max substate values…" dialog box.

- `timing_shell`

  Development window Configure menu "Timing output…" dialog box.

- `simedit_shell`

  Simulation window "Edit substate…" dialog box.

- `simparams_shell`

  Simulation window "Parameters…" dialog box.

- `dialog1_shell`

  Generic one-field dialog box.

- `msgbox_shell`

  Generic message box with "Dismiss" button.

- `confirm_shell`

  Generic confirmation dialog box with "Yes" and "No" buttons.

4.7 Global variables and data structures

4.7.1 Major data structures

- `VIZWIN`

  For each Visualisation window opened, a `VIZWIN` structure is allocated. This holds all the attributes associated with a Visualisation window: Xlib data (pixmap, GC, color-map, etc.), pointer to corresponding shell widget, display type, scaling factors, plane IDs of planes displayed in window, etc.

- VIZWINLISTNODE

  Structure used for plane-to-window mapping linked lists (see `plane2win[]` description).

4.7.2 Major global variables

- `CptCADef cadef`

  CA definition structure used by CARPET parser. Information on CA dimensions, substates and parameters declared in CARPET source is entered into it by the parser during CARPET compilation.

- `unsigned int xyzdims[3]`

  Current dimensions of CA, as defined via the Development window menu item "Configure->Automata Dimensions…".

  If the CA is 1-D or 2-D, then `xyzdims[ZDIM]=1`; if 1-D, `xyzdims[YDIM]=1` also.

- `int nvizwins`
- `VIZWIN *vizwins[]`
- `VIZWINLISTNODE *plane2win[]`
- `int planerefcnt[]`
- `plane_list all_planes`

  These are further discussed in the Visualisation section 8.2.2.

4.7.3 Callback context variables

In order to associate a single callback function with generic dialog boxes such as `filesel_shell` and `dialog1_shell`, global context variables are set before the dialog box is popped-up. These variables preserve state required for the callback function. The following context variables are used:

- `Widget dialog_context`

  Pointer to widget (menu item or button) responsible for popping-up `dialog1_shell`.

- `int substate_context`
  Saved CA substate ID when selecting substate and file.

- `int substatendx_context`
  Saved CA substate index when selecting substate.

- `int disptype_context`
  Saved display type (orthographic, isometric 1-plane, isometric 3-plane) when opening new Visualisation window.

- `int dispplane_context`
  Saved display plane (*x-y*, *x-z* or *y-z*) when opening new Visualisation window.

- `int param_context`
  Saved CA parameter ID when setting parameter.

- `int cell_context[3]`
  Saved cell coordinates when selecting cell.

- `VIZWIN *vizwin_context`
  Saved `VIZWIN` struct pointer when setting Visualisation window colormap.

4.8 List of Functions

4.8.1 Functions in `camelot_stubs.c`

- `void dev_file_open(Widget w, XtPointer client_data, XtPointer xt_call_data)`
  Callback for Development window "File->Open..." menu item.

- `void dev_file_save(Widget w, XtPointer client_data, XtPointer xt_call_data)`
  Callback for Development window "File->Save" menu item.

- `void dev_quit(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for Development window "File->Exit" menu item.

- `void dev_save_and_quit(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for `confirm_shell` dialog "Yes" button.

- `void dev_quit_confirm(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for Development window "Exit" menu item.

- `void dev_edit_cut(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for Development window "Edit->Cut" menu item.

- `void dev_edit_copy(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for Development window "Edit->Copy" menu item.

- `void dev_edit_paste(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for Development window "Edit->Paste" menu item.

- `void dev_config_popup(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for buttons which pop-up the 1-field dialog box (`dialog1_shell`).

- `void dev_config_set(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for dialogs popped-up by Development window "Configure" menu items and some Simulation window "Setup" menu items (`devdims_shell`, `devcc_shell`, `devmpi_shell`, `fold_shell`, `timing_shell`, `minmax_shell` or `dialog1_shell`). Uses `dialog_context` for `dialog1_shell` dialogs.

- `void dev_compile(Widget w, XtPointer client_data, XtPointer`
  `                    xt_call_data)`

  Callback for Development window "Compile" button - calls CARPET parser via `cpt_init()`, `cpt_parse()` and `cpt_finalize()` functions.

  Some support for an external CARPET parser, used in early development of CAMELot remains in the source code (`#ifdef`'d out), but has not been tested recently.

- `void dev_build(Widget w, XtPointer client_data, XtPointer`
  `                    xt_call_data)`

  Callback for Development window "Build" button - runs C compiler.

- `void dev_run(Widget w, XtPointer client_data, XtPointer`
  `                    xt_call_data)`

  Callback for Development window "Run" button - opens Simulation window, initialises sockets, spawns CA Engine, initialises visualisation data.

- `void dev_spawn(Widget textwidget, char *cmdline)`

  Spawn `cmdline` using `popen(3)` and feed `stdout` into XmText widget `textwidget`, keeping cursor at, and showing, end of text. Used by `dev_build()`.

- `void dev_compile_errhndlr(int code, int lineno)`

  Error handler callback for CARPET parser. Calls `cpt_error_message()` to get error message string from CARPET parser, inserts this into a message box XmText widget; also calls `XBell()` and moves the cursor of the Development window XmText widget to the offending line for the first error of a parse only (when global variable `cpt_err_occurred==FALSE`).

- `void dev_set_wintitle(void)`

  Sets window title on Development window, appending `last_carpet_file`. Called by `file_save()` and `file_open()`.

- `int file_open(char *filename, Widget textwidget)`

  Opens file `filename` and reads contents into XmText widget `textwidget`. Also updates global variable `last_carpet_file` and attempts to load configuration file from same directory with same base name as `filename` but with a `.cnf` extension, ignoring errors. Returns -1 on error, otherwise 0.

- `int file_save(char *filename, Widget textwidget, int conf_flag)`
  Writes contents of XmText widget `textwidget` to file `filename`. If `conf_flag` is TRUE, also calls `conf_save()` to save a configuration file to same directory with same base name as `filename` but with a `.cnf` extension. Returns -1 on error, otherwise 0.

- `void sim_go(Widget w, XtPointer client_data, XtPointer`
  `             xt_call_data)`
  Callback for Simulation window "Go" button. Sends EVOLVE request to CA Engine.

- `void sim_loop(Widget w, XtPointer client_data, XtPointer`
  `               xt_call_data)`
  Callback for Simulation window "Loop" button. Sends LOOP request to CA Engine.

- `void sim_pause(Widget w, XtPointer client_data, XtPointer`
  `               xt_call_data)`
  Callback for Simulation window "Pause" button. Sends PAUSE request to CA Engine.

- `void sim_resume(Widget w, XtPointer client_data, XtPointer`
  `                xt_call_data)`
  Callback for Simulation window "Resume" button. Sends RESUME request to CA Engine.

- `void sim_exit(Widget w, XtPointer client_data, XtPointer`
  `              xt_call_data)`
  Callback for Simulation window "Exit" button. Closes all Visualisation windows, sends EXITCODE to CA Engine and closes sockets. Also called from various places to terminate CA Engine.

- `void file_select(Widget w, XtPointer client_data, XtPointer`
  `                 xt_call_data)`
  Callback for buttons which pop-up the file selector dialog (`filesel_shell`).

- `void file_selected(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for file selector dialog (`filesel_shell`).

- `void substate_select(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Pops up the substate selector dialog (`substate_shell`).

- `void substate_selected(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for substate selector dialog (`substate_shell`) and substate editor dialog (`simedit_shell`).

- `void cell_select(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Pops up cell selector dialog (`cell_shell`) with scales set from `xyzdims[]`.

- `void cell_selected(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for cell selector dialog (`cell_shell`).

- `void param_selected(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for parameter editor dialog (`simparams_shell`).

- `void disptype_selected(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for display type selector (`disptype_shell`).

- `void disptype_toggle(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Enable or disable plane selection radiobox according to `disptype_iso3_toggle` value.

- `void popup_msgbox(XmString string)`

  Pops up message box dialog shell (`msgbox_shell`) with XmLabel text set to `string`.

- `void viz_button(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for Visualisation window buttons.

- `void sim_reset_gen_count(void)`

  Set initial Simulation window step number label to " - ".

- `void simedit_init(void)`

  Initialises substate editor dialog (`simedit_shell`) and pops it up.

- `void simparams_init(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Initialises parameter editor dialog (`simparams_shell`) and pops it up.

- `void eng_rx_callback(XtPointer xtp, int *source, XtInputId *xtinid)`

  `XtAppAddInput()` callback for receiving messages from CA Engine.

- `req_code eng_rx_packet(void)`

  Receives packet from CA Engine and handles it appropriately. Returns packet header (`req_code`) or `IGNORED` on error.

- `req_code eng_wait_vispack(void)`

  Block until data received from the CA Engine (or timeout occurs). If packet received, call `eng_rx_packet()` and repeat until it is a `VIS_PACK`. Returns `VIS_PACK` on success or `IGNORED` on failure. Called from `vis_open()` to receive and display initial `VIS_PACK`.

- `void text_search(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for Development window "Find next" button (also called from `dev_config_set()` after "Find" button selected). Searches XmText widget `dev_shell->dev_text1` for text `findtext` (global variable). If called back from `dev_find_button`, start at beginning of text, else if called back from `dev_findnext_button`, start at cursor position. If found, moves cursor to start of text.

- `void text_replace(Widget w, XtPointer client_data, XtPointer xt_call_data)`

  Callback for Development window "Replace text" dialog. Searches XmText widget `dev_shell->dev_text1` for text in XmText widget `devreplace_text1` and replaces all occurrences with text in `devreplace_text2`. Also sets `findtext` to contents of `devreplace_text1`.

- `int conf_load(char *filename)`

  Load Development window configuration `data` (`xyzdims[]`, `nprocs`, `nfolds`, `timing_flag`, `timing_step`, `ccname`, `cflags`, `clibs`, `mpicmd`, `timing_file`) from file `filename`. Sets appropriate widgets to the new values. Returns -1 on error, otherwise 0.

- `int conf_save(char *filename)`

  Save Development window configuration `data` (`xyzdims[]`, `nprocs`, `nfolds`, `timing_flag`, `timing_step`, `ccname`, `cflags`, `clibs`, `mpicmd`, `timing_file`) to file `filename`. Returns -1 on error, otherwise 0.

4.8.2 Functions in `camelot_viz.c`

- `void viz_open(int x, int y, int z)`

  Create a new Visualisation window and corresponding `VIZWIN` struct; tell CA Engine about new plane(s) to visualise.

- `void viz_exit(Widget widget, XtPointer closure, XtPointer call_data)`

  Window manager Delete callback for Visualisation windows (also called from "Close" button callback).

- `void viz_render_plane(VIZWIN *vizwin, int id, unsigned char *data)`

  Renders a plane with ID `id` in a Visualisation window which corresponds to `vizwin` from data pointed to by `data`.

- `void viz_set_def_colmap(VIZWIN *vizwin)`
  Set colormap of window `vizwin` to default palette (red-to-blue spectrum). Sets colormap of both XmDrawingArea and shell widget.

- `void viz_load_colmap(char *filename)`
  Opens palette file `filename` and sets colormap of Visualisation window pointed to by `vizwin_context` according to contents of file.

  Palette files are in ASCII format and consist of 256 lines each containing three space-separated unsigned 16-bit hex numbers specifying the R, G and B values respectively of the palette entry corresponding to the line number (counting from 0). The first line specifies the colour to be used for the background.

- `void viz_draw_colscale(VIZWIN *vizwin)`
  Draws horizontal colour palette bar of height `VIZCOLSCALEHEIGHT` across the bottom of Visualisation window pixmap.

- `void viz_draw_colscale_limits(VIZWIN *vizwin, double min, dou-`
  `ble max)`
  Draw numeric upper and lower limits above colour palette bar in Visualisation window pixmap using colour from middle of colormap. Store limit values, strings and string metrics in `VIZWIN` struct for next call.

- `void spectrum(int idx, XColor *color)`
  Returns a colour in `color` corresponding to the value of `idx` compared to `VIZCOLS` (the number of colours to be used for visualisation). $0 \leq idx < VIZCOLS$. The colour range is a spectrum from blue to red.

4.9 GUI-CA Engine Protocol Requests

All requests to the CA Engine are sent using the macros in Table 3. They are defined in `camelot.h`.

These macros send the request code, call `consume_vis_pack()` to discard outstanding visualisation packets (except for `REQ_EXIT`) and then call the corresponding `req_*()` function in `libcmtguicomms` if present.

Macro	Request code
REQ_SAVE_REQUEST	SAVE_REQUEST
REQ_SET_FOLD	SET_FOLD
REQ_SET_LOAD	SET_LOAD
REQ_VIEW_STATE	VIEW_STATE
REQ_SET_STATE	SET_STATE
REQ_GET_PARAM	GET_PARAM
REQ_SET_PARAM	SET_PARAM
REQ_EVOLVE	EVOLVE
REQ_LOOP	LOOP
REQ_RESUME	RESUME
REQ_TERMINATE	TERMINATE
REQ_PAUSE	PAUSE
REQ_ADD_PLANE	ADD_PLANE
REQ_DEL_PLANE	DEL_PLANE
REQ_PROJ_READ	READ_PROJECT
REQ_PROJ_SAVE	SAVE_PROJECT
REQ_PERIODIC_SAVE	PERIODIC_SAVE
REQ_SET_MINMAX	SET_MINMAX
REQ_EXIT	EXITCODE

Table 3: Correspondence between CAMELot GUI macros and `req_codes`

5. Cellular Automata Engine Implementation

The CA Engine component of the program performs the evolution of the model specified in the CARPET program. Because of the computational intensity of bioremediation models and the inherently parallel characteristics of Cellular Automata, the CA Engine is implemented as a parallel program adhering to the Single Program Multiple Data paradigm. Each process thus created is called a *macrocell* and can apply the transition function of the model locally to a subset of the model, under the assumption that it holds locally all the data that it requires. This suggests the introduction of boundary data which are maintained in neighbouring macrocells and communicated to the process after each evolution.

This communication is implemented using MPI-1 [MPIf 1995], a portable interface for parallel programming. The CA Engine also needs to communicate with the GUI. This communication is executed between one process and the GUI using a purpose-built protocol on top of sockets, as explained in section 7. The process communicating with the GUI is commonly called the *root* process and is selected as the one with rank 0 in the `MPI_COMM_WORLD` MPI Communicator. It coordinates the other processes in order to serve the GUI requests. In this section we discuss the macrocell process implementation and leave the protocol and MPI interaction to section 7.

In addition to the application of the transition function, a *steering* function is available, by which global reductions are performed after each iteration. This is described in section 5.6.3. The system also performs periodic state saves (section 7) and substate visualisations (section 8), as well as timing of the main functions (section 5.7).

5.1 Program Structure

The main CA program component is contained in the file `macrocell.c`. This references global variables, external function and variable declarations in the CARPET generated program and also contains shared and static function prototypes and their code and the `main()` function. The CA Engine functions which implement the communication protocol are also included in the same file but their discussion is deferred until section 7.4. The CA Engine also uses objects defined in the `libcmtcommon` library.

5.1.1 User-Defined Types

Apart from the predefined C types, the following types are used in `macrocell.c`.

5.1.1.1 `CptCell`

The type depends on the definition of the cell in the CARPET program. The code for the struct is written by the parser into the generated header file. The generated type definition for the "Game of Life" example model, which has one substate `life` of type `char`, is as follows.

```
typedef struct _CptCell
{
    char    life ;
}
CptCell;
```

5.1.1.2 `plane` and `plane_list`

These types are used for the visualisation facility on both the GUI and the CA Engine. They are included in `plane.c`. We discuss them in section 8.1.

5.1.1.3 `timer` and `stats`

These are used in conjunction with the statistics output of the CA Engine. Their definitions are included in `macrocell.c`; they are as follows.

```
typedef struct {
    double start;
    double stop;
    double sum;
    double best;
    double worst;
    u_char started;
    unsigned long called;
    char title[TITLE_LENGTH];
} timer;

typedef struct {
    timer func;
    timer vis;
    timer prj;
    timer bound;
```

```
    timer steer;
    timer total;
    int rank;
    unsigned long gens;
    int start_gen;
    u_int period;
    u_char work;
    FILE *outfile;
} stats;
```

The `tmr_code` type is also used in the context of the above structures.

```
typedef enum {
        FUNC = 9999,
        VIS,
        PRJ,
        BOUND,
        STEER,
        TOTAL
} tmr_code;
```

5.1.1.4 `state_dt` and `state_dt_list`

These types are used for the output of the AVS/Express field files, discussed in section 7.5.8. Their definition follows:

```
typedef struct {
    MPI_Datatype data;
    int states;
    int state_ind[NumOfStates];
} state_dt;

typedef struct {
    state_dt statetypes[NumOfStates];
    int many;
} state_dt_list;
```

5.1.2 Functions in `macrocell.c`

5.1.2.1 File I/O Related

```
static int cmt_read (char *, int);
static int cmt_read_global (char *);
static int cmt_read_all (char *);
static int cmt_write (char *, int);
static int cmt_write_global (char *);
static int cmt_write_all (char *, char *);
static int cmt_create_fld (char *, char *);
static int cmt_write_fld (char *, char *, int);
static int check_fs (char *, char *);
int cpt_save (char *);
```

Function `cpt_save()` is used in conjunction with steering (see section 3.3.5 for more).

5.1.2.2 `state_dt` and `state_dt_list` Related

```
static void init_state_dt (state_dt *, MPI_Datatype);
static int add_state (state_dt *, int);
static void init_state_dt_list (state_dt_list *);
static int add_state_dt (state_dt_list *, int);
```

5.1.2.3 Boundary Exchange

```
static int cmt_boundary_copy (CptCell *);
static int cmt_boundary_swap (CptCell *);
static void init_boundaries (void);
```

5.1.2.4 Fold Related

```
static void get_x_line (CptCell *, int, u_char *, int);
static void set_x_line (CptCell *, int, u_char *, int);
static void calc_x_sizes (void);
static void line2fold (const u_char *, u_char *, size_t);
static void fold2line (const u_char *, u_char *, size_t);
static void get_write_ptr (u_char *, u_char *, int, int, int,
                           int, int, int, int);
```

```
static void get_scatter_ptr (u_char **, u_char *, u_char *,
                             int , int);
```

5.1.2.5 Visualisation Functions

```
static int serv_add_plane (void);
static int serv_del_plane (void);
static int tx_vis_pack (cell *, char);
static int serv_set_minmax (void);
static void colour_map (const u_char *, u_char *, int, int,
                        double, double);
static void check_plane (plane *);
static void bcast_plane (plane *, MPI_Comm);
```

5.1.2.6 Protocol Service

```
static int rv (req_code);
static int serv_save_request (void);
static int serv_set_fold (void);
static int serv_set_load (void);
static int serv_view_state (void);
static int serv_set_state (void);
static int serv_set_param (void);
static int serv_terminate (void);
static int serv_proj_read (void);
static int serv_proj_save (void);
static int serv_periodic_save (void);
static int send_ack (int);
```

5.1.2.7 CA Execution Function

```
static int run (void);
```

5.1.2.8 Random Number Generators

```
void cpt_randomize (void);
void cpt_srandom (unsigned int);
```

These functions provide pseudo-random number generators and are the only shared (public) functions defined in `macrocell.c`.

5.1.2.9 Statistics Output

```
static void init_tmr (timer *, const char *);
static int start_tmr (timer *);
static int stop_tmr (timer *);
static void print_tmr (const timer *, FILE *);


static void init_sts (stats *, int, u_char, u_int, const char *);
static void reset_stats (stats *);
static int start_one_timer_sts (stats *, tmr_code, int);
static int stop_one_timer_sts (stats *, tmr_code, int);
static void print_sts (const stats *);
static void close_file_sts (const stats *);
```

5.1.2.10 Other Functions

```
static void check_pos (int *);
static int get_val_size (const int *);
```

5.1.3 External Function Prototypes

```
extern void cpt_hook_init (void);
extern void cpt_func (CptCell *, CptCell *);
extern void cpt_set_state (CptCell *, int, void *, int);
extern void cpt_get_state (CptCell *, int, void *, int);
extern void cpt_mpi_type_cell (MPI_Datatype *);
extern void cpt_hook_finalize (void);
extern int cpt_thresh (CptCell *);
extern req_code cpt_steering (CptCell *);
```

5.1.4 External Variables

```
extern float cpt_globpar[NumOfGlobPar]; /* CARPET parameters */
extern const size_t cpt_state_size[NumOfStates];
                                        /* Substate bytesizes */
```

```
extern MPI_Datatype cpt_state_mpidt[NumOfStates+1];
                                    /* Substate MPI Dtypes */
extern bool_t (*cpt_state_XDRfn[NumOfStates])();
                                    /* Substate XDR function*/
extern const int cpt_determin;          /* Deterministic flag */
extern const char *cpt_state_name[NumOfStates];
                                    /* Names of states */
```

5.1.5 Global Variables

```
CptCell *ca1;                   /* 1st copy of Cellular Automata */
CptCell *ca2;                   /* 2nd copy of Cellular Automata */
int cpt_dimx, cpt_dimy, cpt_dimz; /* Local CA dim sizes */
int cpt_x, cpt_y, cpt_z;        /* Current coordinates, X, Y, Z
                                   Must be global for Get[XYZ] */
char *out_basename = NULL;       /* Root of periodic save filename */
char *out_dirname = NULL;        /* Path of periodic save filename */
int save_step = INT_MAX;        /* Period of saves */
int num_gens;                    /* Remaining generations to run */
int cpt_generation = 0;      /* Must be initialised for vis_list */
char in_steering = 0;           /* Flag for steering statement */


int cmtWorldSize, cmtWorldRank;
                                /* MPI world size and local rank */
int cmtPrevRank, cmtNextRank; /* Ranks of neighbour macrocells */
MPI_Comm cmtCommCommand, cmtCommBoundary;
                                /* MPI contexts: data & control */
MPI_Datatype cmtBoundaryType; /* MPI datatype for boundary data */
int prot_sockfd = 0, vis_sockfd = 0;
                                /* protocol and visualisation socket
                                     fd, global for comm abstraction
                                     independence */


list vis_list;                  /* List of visualised planes */
plane_list all_planes;      /* List of all planes in the Engine */
int alt_intnl_bound[2][NFOLDS];
                                /* Left/Right internal boundary
                                     altered flag for each strip */
```

```
int alt_strip[NFOLDS];          /* Internal cells altered flag */
int active_strip[NFOLDS];       /* Active Strip flag */
int manual_folds = 0;           /* Flag to examine (in)active folds */
double minmax[NumOfStates][2];  /* Minimum/Maximum value for
                                   each state. */


char auto_map[NumOfStates];     /* Flags for automatic colour map */
stats statistics;               /* Statistics for timers */
state_dt_list *dt_list = 0;     /* state_dt_list for AVS files */


#ifdef EVEN_DECOMP⁹
#else
int CPT_DIMX [CPT_NPROCS];            /* Actual x data in process */
int CPT_F_X [CPT_NPROCS][NFOLDS];    /* Actual x data per Strip */
int CPT_S_X [CPT_NPROCS][NFOLDS];    /* Total Strip x-size */
int first_small_strips_ind[CPT_NPROCS]; /* Index of first strip of
                                          process to have smaller size */
#endif
```

5.2 Data Handling

5.2.1 Internal Representation

Each cell can be thought of as a 3-D (*x, y, z*) triplet of co-ordinates with an associated set of substate values. In [Telford et al. 1998] it was decided that cells would be represented as C structs. The CA is represented as an array of such cells. If the program is run on more than one process, each process contains a fraction of the model data. The decomposition of the data to processing elements is discussed in section 5.4. Because of the way the CA execution function works, each process maintains two such arrays, one to contain the data of the previous iteration and another where the output of the transition function is written. After the transition function has been applied to all the cells of a macrocell, the read copy is updated. This is necessary for the correct execution of the program, despite the fact that the read and write copies are toggled after each step. A good example of a program that fails is one that updates the cells with odd *x*-coefficient on the odd generations and the even ones on the even generations.

---

[9] The EVEN_DECOMP macro definition is explained in section 3.2.1.1.3.3.

The cells are accessed through the following macro, defined in the file `cpt_ccdefs.h`.

```
#define CA_REF( ca, z, y, x ) ((ca) + (z)*CPT_Y*CPT_X + (y)*CPT_X + (x))
```

As we discuss later in the section, duplicate, boundary data are incorporated in the real CA data to allow the execution of the transition function. `CPT_X` and `CPT_Y` are the total dimensions across the *x* and *y* axis of each process respectively, including the real and replicated data. The above macro implies that the *x*-axis is "moving" fastest when accessing the cells and because the decomposition is done across the *x*-axis, boundary data are located between two consecutive *x*-lines in the dataset of each process. Therefore, the data are fragmented thus reducing the expected benefit from processor read-ahead and caching optimisations when applying the transition function [Telford et al. 1998]. It should be noted that whether performance could be improved by running the *z*-axis fastest depends on the model size and the decomposition, since boundary data are replicated across all axes. Moreover, this approach effects non-contiguous boundaries, thus necessitating the introduction of MPI boundary datatypes which contain non-contiguous data. This also possibly affects the performance of the CA Engine, as discussed in [Kavoussanakis et al. 1999], but no conclusive evidence has been found during the profiling of the software.

5.2.2 Data I/O

CAMELot supports state initialisation from files as well as saving the state to files. Files output to disk can be used for state initialisation at a later stage without any transformation. Data transfer occurs also between the CA Engine and the GUI both in order to control the execution and to provide visualisation of the states.

Starting from release 1.2 of the software, the XDR data representation standard is supported for file I/O *only*. The use of XDR for read-related operations is discerned from that for write-related operations, as they are controlled by means of different C pre-processor definitions in macrocell.c (`NO_XDR_READ` and `NO_XDR_WRITE` respectively). This allows the user to use the system as a filter to translate old binary files to XDR-based ones by defining the `NO_XDR_READ` flag, as shown in section 3.2.1.1.3.1. In order to achieve this, the `cpt_state_XDRfn[]` pointer-to-function array is generated from the parser and defined in the C file. This provides the appropriate XDR primitives to translate the substate elements to their corresponding external representation.

File I/O is effected with one call, both when XDR is used and otherwise. The corresponding functions (`cmt_read()` and `cmt_write()`) use adequately large buffers to fit the

data, as discussed in section 7.5.3.1. Most of the times I/O to the socket is also done with one call.

5.3 Process Placement

The user defines the number of processes from the Configure Menu of the Development Window. These processes are then arranged in a 1-D, periodic Cartesian topology represented by MPI Communicator `cmtCommBoundary`, to enable boundary exchange. The physical allocation of processes to processing elements is hidden by the system. As far as the programmer is concerned, each process has a known couple of neighbours, `cmtNextRank` and `cmtPrevRank`, identified by their rank (for process *i* in a *n*-process system, the previous has rank (*i*-1) mod *n*, the next has rank (*i*+1) mod *n*). An additional, unordered Communicator, `cmtCommCommand`, is also created for controlling the processes. The rank of each process, `cmtWorldRank`, as well as the total size of the system, `cmtWorldSize`, are stored as global variables in each process. They are acquired by means of standard MPI calls. The Cartesian arrangement of processes and the creation of the two Communicators are also achieved using MPI calls.

5.4 Data Decomposition

The CABOTO project introduced a form of block-cyclic decomposition aiming to reduce load imbalance [Spezzano et al. 1995]. The idea, which was also implemented in CAME-Lot, was to split the model virtually in a number of *folds* and then assign equal parts (if this is possible) of each fold to each of the processes (Figure 27). This can lead to load balancing under the condition that the resulting granules (further referred to as *strips*) are fine enough to ensure that the uneven load distribution across folds statistically is insignificant across processes. It should be noted though, that the numbers of folds and processes should be chosen with caution during the executable build phase, since the more the strips, the bigger the communication overhead among the processing elements.

The model is decomposed across the *x*-axis; this suggests that, in order to utilise the available resources, 1-D models should be considered as *x* lines and 2-D models should be viewed as *x*-*y* planes. The fold and process numbers are defined by the user through the Configure Menu of the Development Window and they are passed to the CA program as compiler line arguments using the –D option. The radius of the neighbouring cells is defined in the CARPET program through the `radius` statement (see section 3.3.15).
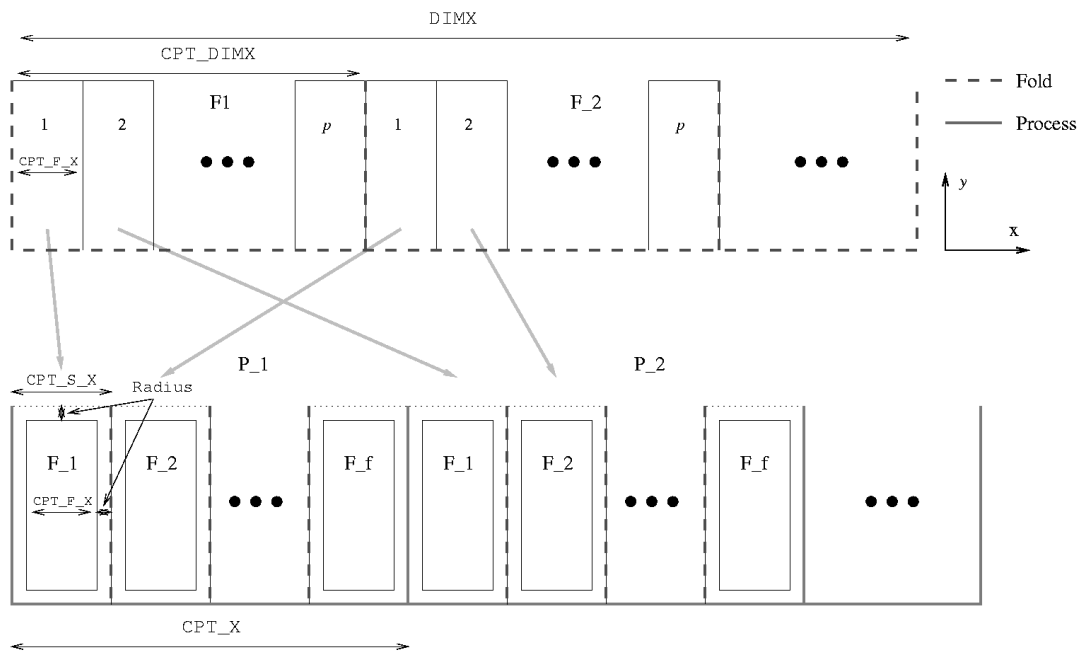
Figure 27: Block-Cyclic Decomposition in CAMELot. The figure and the notation assume even decomposition.

5.4.1 Uneven Decomposition

For CAMELot releases prior to 1.3 it was necessary for the product of the number of folds times the number of processes to divide the total *x*-size of the model. This condition was relaxed in CAMELot 1.3. It should be noted that the former implementation is more efficient since it carries less overhead, whereas the latter is more general. As a result of uneven decomposition, some processes may have their *x*-dimension larger than others by one. Similarly, some strips in a process may have their *x*-dimension larger by one.

The definition of the *x*-sizes of the processes and strips takes place in function `calc_x_sizes()`. The strategy for processes is that the root process will have at most as many elements as the others, on the grounds that it has extra workload because of the communication duties. On the contrary, strips are allocated extra elements in their *x*-dimension starting with strip 0.

There are two useful corollaries from the above discussion:

- In each process there is a strip which has the lowest index of those with less elements than the others and the index of small strips range from this lowest index to `NFOLDS-1`.

This index for each process is stored in the file `macrocell.c` in the global array `first_small_strip_ind[CPT_NPROCS]`. In the case of even decomposition, this index defaults to 0.

- The size of the smallest strip is the size of strip indexed `NFOLDS-1` in the root process.

Example: Suppose that the total *x*-dimension of a 1-D model is 50, divided in 3 processes and 3 folds. Processes 1 and 2 will have 17 elements each, whereas process 0 will have 16. In processes 1 and 2 strips 0 and 1 will have 6 elements each, whereas strip 2 will have 5. In process 0 strip 0 will have 6 elements and strips 1 and 2 will have 5 each. Also, the following is true for the per-process array of indices to small strips.

$$first\_small\_strip\_ind[3] = \{1, 2, 2\}$$

`CPT_NPROCS`	Number of processes
`NFOLDS`	Number of folds
`DIMX`	Total number of data on the *x*-axis of the model
`DIMY`	Total number of data on the *y*-axis of the model
`DIMZ`	Total number of data on the *z*-axis of the model
`CPT_DIMX`	Number of actual data on the *x*-axis of a process
`CPT_F_X`	Number of actual data on the *x*-axis of a strip
`CPT_S_X`	Total number of data on the *x*-axis of a strip (including boundary duplicates)
`CPT_X`	Total number of data on the *x*-axis of a process
`CPT_Y`	Total number of data on the *y*-axis of a process
`CPT_Z`	Total number of data on the *z*-axis of a process

Table 4: Model Definition Notation

5.4.2 Notation

CAMELot supports two implementations for the data decomposition depending on whether this is even or not. The code selected for each case is controlled by the C pre-processor definition `EVEN_DECOMP`. If this is defined, then the program assumes that the product of the number of folds times the number of processes divides the total *x*-size of the model. However if the assumption is false, the program exits with a warning message returning -1.

Otherwise, and this is the default behaviour, the program assumes uneven decomposition of data to processes.

The constants (or macros) in Table 4 define the model. The type of each of the definitions in this Table differs according to the definition of the EVEN_DECOMP macro. This is shown in Table 5 and Table 6 below. Note the difference in the definitions of CPT_DIMX, CPT_F_X and CPT_S_X, in the case of even decomposition they are straightforwardly calculated and stored as macros in the cpt_ccdefs.h header file, otherwise they are defined as arrays stored in macrocell.c and calculated by the function calc_x_sizes().

```
#define CPT_DIMX (DIMX/CPT_NPROCS)    /* Actual x data in each process */
#define CPT_F_X (CPT_DIMX/NFOLDS)     /* Actual x data in each Strip */
#define CPT_S_X (CPT_F_X+(2*Radius))  /* Total Strip x-size */
#define CPT_X (CPT_S_X*NFOLDS)        /* Total process x-size */
#define CPT_Y ((DIMY)+(2*Radius))     /* Total process y-size */
#define CPT_Z ((DIMZ)+(2*Radius))     /* Total process z-size */
```

Table 5: CA Engine Size Definitions (EVEN_DECOMP defined)

```
int   CPT_DIMX[CPT_NPROCS];          /* Actual x data in process */
int   CPT_F_X[CPT_NPROCS][NFOLDS];   /* Actual x data per strip */
int   CPT_S_X[CPT_NPROCS][NFOLDS];   /* Total Strip x-sizes */
#define CPT_X (2*Radius*NFOLDS+CPT_DIMX[cmtWorldRank])
#define CPT_Y ((DIMY)+(2*Radius))
#define CPT_Z ((DIMZ)+(2*Radius))
```

Table 6: CA Engine Size Definitions (EVEN_DECOMP undefined)

5.5 Boundary Replication

Data decomposition effects the introduction of duplicate boundary data. The reason is that splitting the model into strips and allocating contiguous strips to different processes causes some cells to lose immediate neighbours. Given that the strips divide the model across the *x*-axis, the cells which are located in radius distance from the *x*-edges of the strips have

lost their neighbours lying outside the strip and they are thus unable to execute the CA evolution rule.

Moreover, all cells within `Radius` distance from any edge have no defined neighbours outside the model domain. In order to remedy this, we implement cyclic boundaries. This means that data are "wrapped round" in each dimension, so that cells at one edge are neighbours of cells at the opposite edge in the same dimension. This ensures cyclic interaction and execution of cells. The effect to the topology of the model is the following.

- 1-dimensional models are effectively circular;
- 2-dimensional models have the shape of the surface of a torus;
- 3-dimensional models are shaped as a 3-D torus.

The above approach suggests four types of halo:

- before the first and after the last real element (z-axis);
- between planes (y-axis);
- between lines (x-axis);
- between strips (folded data).

The first three haloes conceptually form a shell around the model, whereas the last increases its x dimension. This is depicted in Figure 28. We will discuss the effect of the haloes on the internal representation of the CA.

The first type of halo consists of `Radius` planes of size `CPT_X*CPT_Y` on each z-side of the CA model. It is implemented by prefixing and postfixing the data with contiguous planes of halo data. The plane halo is `Radius` lines of size `CPT_X` on each y-side of the model. It is implemented by adding lines of haloes between planes of data. Similarly, there is a `Radius` sized halo introduced on either side of the model, corresponding to the line halo.
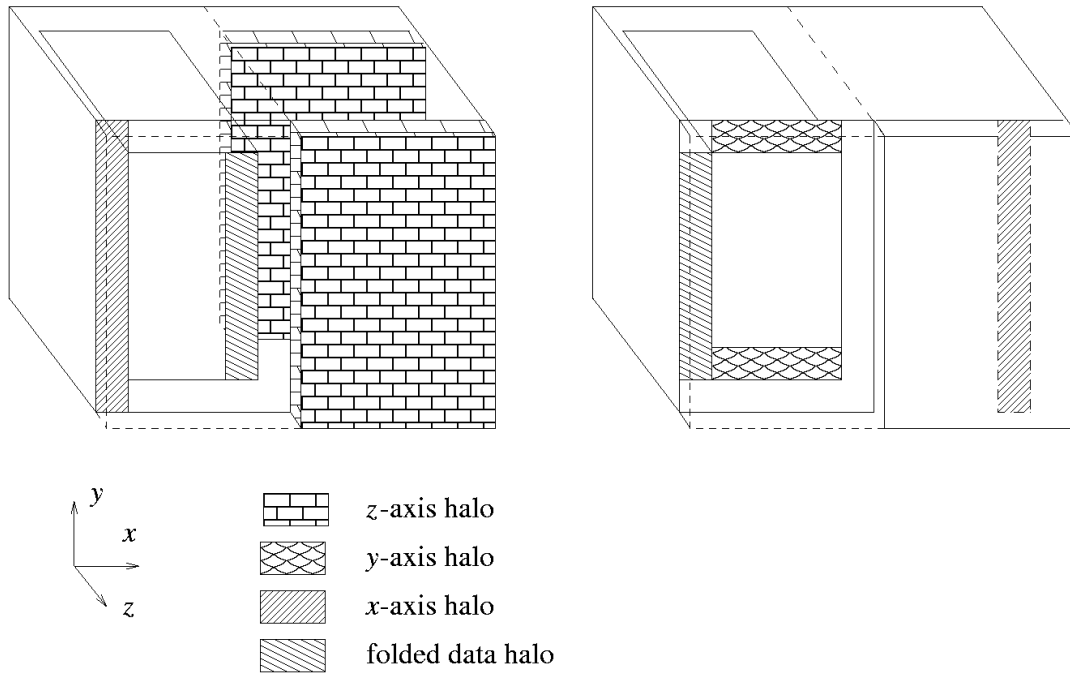
Figure 28: Halo replication: A 2-fold, 2-process decomposition is shown.

The above discussion suggests two different kinds of boundary replication. Across the *y* and *z* axes the data can be replicated internally in macrocells. We call this a *boundary copy*. Across strips (this effects *x*-axis halo replication as well) the data must be exchanged between consecutive processes using MPI. We call this *boundary swap*. Boundary data are copied across axes and swapped between strips after the execution of a step.

5.5.1 Boundary Copy

This is achieved with the following function:

```
static int cmt_boundary_copy (CptCell *ca)
```

This function replicates the local boundary data across the non-distributed dimensions. It is dependent on the cell access macro. It returns zero on completion. Note that this function only performs the boundary copy on one CA array copy.

The *z*-axis (slowest) boundary exchange, is performed with two `memcpy` calls, each copying `Radius*CPT_Y*CPT_Z` elements, i.e. `Radius` *x*-planes. For the *y*-axis copy it loops over *z* with two `memcpy` calls in each iteration. Each `memcpy` handles `Radius*CPT_X` elements, i.e. `Radius` *x*-lines.

5.5.2 *Boundary Swap*

This is done with the following function:

```
static int cmt_boundary_swap (CptCell *ca)
```

This function uses MPI to exchange *x*-axis boundary data between the strips. If the automatic inactive strip detection mechanism is activated, it also exchanges information about the activity of the internal boundaries of its neighbouring cells so as to determine its own activity. The implementation depends on the cell access method (`CA_REF`) and only replicates the data for one CA array copy passed to it as an argument.

In order to define the receiver and the sender of messages we use the global variables `cmtPrevRank` and `cmtNextRank` returned from the call of the Cartesian Topology creation functions of MPI, `MPI_Cart_create()` and `MPI_Cart_shift()`. We thus define a global MPI Communicator, `cmtCommBoundary`, used for boundary swapping. We also define a derived datatype, `cmtBoundaryType`, for the boundaries to be exchanged. This is a vector datatype created using the `MPI_Type_vector()` function of MPI. It allows reference to the stridden boundary by specifying only the starting point of the data to be received or sent. It consists of `CPT_Z*CPT_Y` blocks cells, each of which has length `Radius`; the stride between consecutive blocks is `CPT_X`. This communicator is global to our program.

In order to exchange activity and boundary data, two similar blocks of communication primitives have been developed which differ only in the data they exchange. Each of the blocks contains two loops over the number of strips in the process, one to receive and one to send the data. The activity data received from the previous neighbour for strip `i` are stored in `prev_active[i]` and the data from the next in `next_active[i]`. The corresponding tags for the messages are `2*i` and `2*i+1`.

The definition of the index for the data to be sent is less straightforward. The general rule for sending data to the next process is to send the right internal boundary to the strip of the *same* rank on the right of the sender. From the above we see that the receiver strip `i` waits for a message tagged `2*i`. This means that the message to the next process contains `alt_intl_bound[1][i]` and is tagged `2*i`. The former does not hold for process `cmtWorldSize-1` whose next neighbour is process `0`, since this must send the data of the previous rank to process `0` (see Figure 27) in order to implement cyclic boundaries. Thus,

the data sent to process 0 are `alt_intnl_bound[1][(i-1)%NFOLDS]`[10]. Similarly, the data sent to the previous process are stored in `alt_intnl_bound[0][i]` except for process 0 which sends the data stored in `alt_intnl_bound[0][(i+1)%NFOLDS]`. In both cases the message is tagged `2*i+1`.

The above hold for the actual boundary swapping block as well. Reception from the previous strip starts from `CA_REF(ca,0,0,i*CPT_S_X)`, and is tagged `2*i`; reception from the next strip starts at `CA_REF(ca,0,0,(i+1)*CPT_S_X-Radius)`, tagged `2*i+1`. Data to the next process are tagged `2*i` and start at `CA_REF(ca,0,0,CPT_F_X+i*CPT_S_X)` except if the sender is process `cmtWorldSize-1` which sends data starting at `CA_REF(ca,0,0,CPT_F_X+((i-1)%NFOLDS)*CPT_S_X)`. Data to the previous process start at `CA_REF(ca,0,0, i*CPT_S_X+Radius)` except for process 0 which sends data that start at `CA_REF(ca,0,0,((i+1)%NFOLDS)*CPT_S_X+Radius)`. The execution of this block does not start until all the activity data have been received from the previous process; when the receives from the previous and next neighbour are issued, `active_strip[i]` is updated according to `prev_active[i]` and `next_active[i]`.

Although immediate sends and receives have been used for the implementation of boundary swaps, the current implementation does not execute the transition function for internal cells while boundaries are exchanged, as suggested in [Telford et al. 1998]. Efficient boundary swapping, taking into account whether each strip is active or not, has not been implemented either.

5.5.3 *Function* `init_boundaries()`

In order to facilitate boundary exchange when the system is restarted we implemented the following function.

```
static void init_boundaries (void)
```

This function assumes that the system has been brought to a new state and cancels all of the strip activity data previously defined by the automatic mechanism. It thus sets `active_strip[]` and `alt_intnl_bound[][]` for all of the strips before calling `cmt_boundary_copy()` and `cmt_boundary_swap()` on *both* copies of the CA array. It then cancels `alt_intnl_bound[][]` and `alt_strip[]`.

---

[10] In fact the code reads `alt_intnl_bound[1][(i-1+NFOLDS)%NFOLDS]` to ensure correctness of the modulus operator.

N.B.: This function must only be used when the system is brought to a new state, as it affects the strip activity variables. Nonetheless, if called at any point, it does not affect the correct execution of a deterministic program.

5.6 Transition Function Execution

5.6.1 CA Engine States

The CA Engine can be in any of the following states:

- Running;
- Serving Protocol Requests;
- Paused;
- Stopped.

Protocol requests (including `PAUSE` and `TERMINATE` which effect the two last states) can be received at any point, but they are only handled before a CA Engine iteration. These are implemented with static variables in the `rv()` function, discussed in section 7.5.2.

5.6.2 Automatic Inactive Strip Detection

CAMELot contains an *automatic inactive strip detection* mechanism, used to isolate inactive regions and avoid applying the function to idle strips. The block cyclic decomposition suggests that load imbalance emanating from this strategy will be insignificant in the general case, given that contiguous areas of the model are transparently distributed to processes. Automatic inactive strip detection can be disabled by manually choosing a set of active folds as described in section 3.2.2.1. The finest grain in this case is the fold, which is generally larger than the strip, and the strategy is error prone as it depends on the user's vigilance. Moreover, manual fold selection cannot isolate inactive regions located in the middle of the model even if the granularity suffices, because the active range defined is continuous. The implementers do not recommend manual fold selection. We will discuss the implementation of the automatic inactive strip detection mechanism in 5.6.3.1.

5.6.3 Function `run( )`

The `run()` function of the program loops over the requested number of CA Engine generations and applies the update function to all the cells. It also executes the steering function, transmits the current generation number to the GUI and initialises the visualisation and periodic saves which are due in this iteration. Before each iteration the root process polls

the socket for pending messages by means of a `select(3C)` call. If a message is present it broadcasts it to the other processes. The control at this point is passed to the `rv()` function, discussed in section 7.5.2. The processes synchronise loosely by means of this `MPI_Bcast` call as they exit this loop and enter running mode when all pending requests have been served.

At the end of the iteration the boundaries are replicated on the write copy of the CA array and this is then copied to the read copy by means of a `memcpy` call. The *steering* function is applied to one copy of the automaton only. This does not affect the execution of the model, because the steering function cannot alter the state of cells. Immediately before calling the external steering function `cpt_steering()` we set the global variable `in_steering` so as to enable the execution of the steering related functions. We also synchronise the processes so as to avoid race conditions in the update of global parameters. The variable `in_steering` is cleared immediately after exiting the steering function to disable access to the steering related functions. We discuss the steering function later in this section.

After the steering function has been executed, the visualisation list is checked for planes waiting to be visualised; if there are any, `tx_vis_pack()` is called with the `force` argument set to 0. After each visualisation, the `generation` member of the cell is set for the next visualisation, and the cell is inserted in the correct position in the visualisation list, using the function `reorder()`, discussed with the `list` functions, in section 8.3.5. Then the current generation number is written to `vis_sockfd`. Finally, periodic project saves are performed by means of the `cmt_write_all()` function if the incremented generation is divided by the `save_step`. If the generation run is the last one requested, and periodic saves are enabled, but no periodic save has occurred in the current step, then `cmt_write_all()` is called to save the final configuration of the CA.

The implementation of function `run()` is shown in pseudocode in Table 7.

```
for (num_gens) {
    do {
        get_request();
    } while (no_request || stop);

    if (stop)
        return;

    cp = CA_REF (ca, Radius, Radius, 0);
```

```
    for (cpt_z) {
        for (cpt_y) {
            for (strip) {
                cp += Radius;
                for (x) {
                    calculate (cpt_x);
                    update(cell);
                    if (x < l_bound)
                        check_l_intnl_bound()
                    else if (x > r_bound)
                        check_r_intnl_bound()
                    else
                        check_intnl_strip();
                    cp++;
                }    /* End for (x) */
                cp += Radius;
            } /* End for (strip) */
        } /* End for (cpt_y) */
        cp += 2*Radius*CPT_X;
    }    /* End for (cpt_z) */

    update_boundaries(ca);
    update_copies (ca);

    synchronise_procs;
    cpt_steering (ca);

    if (visualisation_due)
        tx_vis_pack();
    if (configuration_save_due)
        cmt_write_all();

    write_gen_no();

}    /* End for (num_gens) */
```

```
if (configuration_not_just_saved)
      cmt_write_all();
```

Table 7: Function `run()` in pseudocode

5.6.3.1 Application of the Transition Function

The transition function is applied to all the cells of a process without any interruption. The processes communicate again for the necessary interaction for boundary swapping. The next step is to identify which of the two pointers to the CA represents the read and write copy by means of the parity of the CA generation and then the function loops over the strips skipping the haloes and applies the transition function `cpt_func()` to its cells.

During this phase the cells in strips are examined in order to decide their activity status. Initially, all the strips are considered active. The system attempts automatic inactive strip detection under the condition that the user has characterised the transition function as deterministic through the CARPET statement `deterministic` [Telford et al. 1998]. After the transition function has been applied to a cell, the cell is checked against `cpt_thresh()` and the previous values of all its substates. If a substate has changed and `cpt_thresh()` returns false, then the whole strip is characterised as active for the next generation and the check is not performed for any other cells of the strip. Even if all the cells of a strip are classed as inactive, the strip is not considered inactive unless the boundary cells to be received by each of the neighbours are also inactive.

In order to achieve the above we virtually split the strips in three components, a block of internal cells and two blocks of internal boundaries on either of its *x* sides. The internal boundaries are part of the strip's cells but they are of special interest as they are communicated to the strips lying on their external sides. Their dimensions are the same as those of the boundary data. During the boundary swap the processes exchange information with their neighbours about the activity status of the incoming boundaries and combine the results with those emanating from the internal cell check to decide on the activity status of their strips.

Haloes are skipped as follows: before the function enters the loop the first *z* and *y* haloes are skipped. In the first strip the *x* halo is skipped and then the application of the function begins. If the strip is idle, the pointers are advanced by `CPT_F_X`, which is the *x*-size of a strip; otherwise, for each of the `CPT_F_X` applications of the function the pointers are advanced by 1 position. At the end of the strip line traversal, the pointers are advanced again

by an *x* halo. This leaves the pointers at the beginning of the first *x* halo of the next strip. After the end of the plane (and the *y* iteration of the loop) the pointers are incremented by `2*Radius*CPT_X`, which skips both the *y* halo at the end of the current plane and that of the next plane.

In addition to the above, the position of the cell updated is recorded by means of the `cpt_x`, `cpt_y` and `cpt_z` global variables. This follows the convention that the co-ordinate of the first cell in each dimension is 1. We also take into account the decomposition of cells in folds and processes so as to enable correct `cpt_x` calculation. The user can access the values of `cpt_x`, `cpt_y` and `cpt_z` in their program using respectively the `Get_X`, `Get_Y` and `Get_Z` CARPET statements (see section 3.3.10), thus enabling position-dependent update functions.

5.6.3.2 Calling the Steering Function

After the two CA copies have been updated, the steering function is being called. This is external to `macrocell.c`. Nonetheless, it is important for the program to set the global variable `in_steering` before calling the steering function and clear it after exiting it. This is external to the CARPET-generated C file, and allows the global reduction functions to be executed. We chose to control this flag from `macrocell.c`, rather than the generated `cpt_steering()` function to avoid possible problems if the CARPET program contains a `return` call inside the `steering` statement. The processes synchronise by means of the MPI `MPI_Barrier` call before executing the steering function, so as to avoid possible race conditions.

5.7 Timing

From release 1.1 of the software onwards, the basic functions of the CA Engine are timed. The functions timed are as follows:

- Transition function;
- Visualisation;
- Project save;
- Boundary replication;
- Steering.

The user can define how often the results are output (see section 3.2 for more details on this). Regardless of this setting, the execution is timed in each step. For each of the functions timed, the following features are monitored and reported:

- The number of calls;
- The total time taken by this function;
- The best and worst time recorded for this function.

The sum of the above times is also reported. Additionally, a timer instance collects statistics for the duration of the *period* of each run. The format of the output is seen in Table 8.

5.7.1 Strategy for Timing the Functions

There are several ways in which a function can be timed. In this section we describe the strategy used for each of the functions.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Process: 0                                                  Generations:

                         Calls            Time            Best            Worst
-----------------------------------------------------------------------------------
Update Function :
Steering        :
Boundary Comm   :
Visualisation   :
Periodic Save   :

Sum             :
Total Execution Time:
```

Table 8: Output of Timing Statistics

5.7.1.1 Transition Function

The transition function is timed inside function `run()`. We start the corresponding timer before entering the nested loop traversing the elements of the model and stop it immediately after exiting it. This provides a *per-PE* granularity. As mentioned in section 5.6.3, after each iteration of the CA Engine, the read copy of the CA is updated by means of a `memcpy` call. Note that the time taken by this call is *not* accounted for by the update function timer, yet it appears in the total timer discussed in section 5.7.1.6 below.

Our initial implementation timed each call to function `cpt_func()`, but the overhead was unacceptable. More specifically, removing the timing functions in a trivial 1000x1000 model running in 1 processor, 1 fold for 10 steps yielded a 70% performance benefit.

5.7.1.2 Visualisation

The visualisation timer is started when entering function `tx_vis_pack()` (see section 8.6.2.1) and stopped before exiting it.

5.7.1.3 Project Save

The project save timer is associated with function `cmt_write_all()`. This allows the timing of the project saves even when the program is running in batch mode (see section 0). This function initialises the writing of all the configuration related files, as described in section 7.5.8.1.

5.7.1.4 Boundary Replication

The start and stop calls for this timer enclose calls to `cmt_boundary_copy()` and `cmt_boundary_swap()`. These two functions are always called together.

5.7.1.5 Steering

The timer is started after explicitly synchronising the processes for the execution of the steering statement and stopped immediately after it has been executed. The time taken for the synchronisation of the processes is *not* taken into account.

5.7.1.6 Total Time

This timer is started when entering the `run()` function and it is kept running while the CA Engine is running. It stops when `rv()` is called to serve user requests (see section 7.5.2) and it is restarted when `rv()` returns. The timer is running when `print_all_stats()`, discussed in section 5.7.2.2, is called but it is stopped and restarted so as to allow for correct statistics gathering in process 0.

5.7.2 Structures and Functions

5.7.2.1 Structure `timer`

We designed the structure `timer` which contains all the necessary data for each of the timed functions.

```
typedef struct {
    double start;
    double stop;
    double sum;
    double best;
    double worst;
    u_char started;
    unsigned long called;
    char title[TITLE_LENGTH];
} timer;
```

The associated functions are as follows:

- `static void init_tmr (timer *tp, const char *title)`
  This initialises a `timer` struct. It assumes that memory has previously been allocated for it. All the members are set to 0, with the exception of `title` which takes the value of the argument[11] and `best` which is set to `DBL_MAX`.

- `static int start_tmr (timer *tp)`
  This starts the `timer` pointed to by `tp`. It increments `called`, sets `started` and assigns to `start` the value returned by `MPI_Wtime()`.

  It returns 0 if the timer was already started, or 1 otherwise.

- `static int stop_tmr (timer *tp)`
  This stops the `timer` pointed to by `tp`. It clears `started` and assigns to `stop` the value returned by `MPI_Wtime()`. It also adds the time between `stopped` and `started` to `sum` and checks if the current record is a `best` and/or `worst` time.

---

[11] If the `title` argument equals `NULL`, the `title` member is not set. The function in this case is used to reset the members of the structure.

This returns 0 if the timer was already stopped, or 1 otherwise.

- `static void print_tmr (timer *tp, FILE *f)`

    This checks if `tp` is `started`, in which case it prints a warning. It then prints the `ti-tle`, followed by one tab and `called` followed by one or two tabs according to its length. It then prints the `sum`, `best` and `worst` separated by a tab character and finishes with a newline character. All the output is in one line.

5.7.2.2 Structure `stats`

This structure is a collection of timers. It includes a pointer to a `FILE` variable which identifies the file where the data are written. It encapsulates various characteristics of the "instance", including a flag indicating whether statistics are taken (`work`), the `period` of outputting the statistics, the number of generations for which statistics are produced (`gens`) and others, as seen below.

```
typedef struct {
    timer func;
    timer vis;
    timer prj;
    timer bound;
    timer steer;
    timer total;
    int rank;
    unsigned long gens;
    int start_gen;
    u_int period;
    u_char work;
    FILE *outfile;
} stats;
```

The associated functions are as follows:

- `static void init_sts (stats *stp, int rank, u_char work,`
                         `char *fname)`

    This initialises a `stats` structure. The `timer` members are initialised with the titles as set inside the function (not passed as a set of arguments). The `outfile` member is

opened in process 0 and set to NULL in all other processes. The rest are straightforward.

- `static void reset_sts (stats *stp)`
  This is applied only to structures where the `work` flag is set. It is used by function `print_all_stats()` discussed below. All the `timer` members are restarted with the titles argument set to NULL (see function `init_tmr()` above) and `gens` is set to 0.

- `static int start_one_timer_sts (stats *stp, tmr_code tmr,`
  `int gen)`
  This starts the `timer` denoted by `tmr`. If `tmr` is not TOTAL, it calls the corresponding `start_tmr()` call and returns what that returns. If it is TOTAL it also checks and sets the `start_gen` member of the `statistics` structure to `gen`.

  It returns 0 if the `statistics` entity does not `work` or if the request is ignored (see the discussion of `start_tmr()` above), a negative value if the arguments are unacceptable or 1 otherwise (successful termination).

- `static int stop_one_timer_sts (stats *stp, tmr_code tmr,`
  `int gen)`
  This starts the `timer` denoted by `tmr`. If `tmr` is not TOTAL, it calls the corresponding `stop_tmr()` call and returns what that returns. If it is TOTAL it also checks and sets its `gens` member to `gen-stp->start_gen`.

  It returns 0 if the `statistics` entity does not `work` or if the request is ignored (see the discussion of `stop_tmr()` above), a negative value if the arguments are unacceptable or 1 otherwise (successful termination).

- `static void print_sts (const stats *stp)`
  This prints all the members of the `stats` structure. It produces the output shown in Table 8. The Sum field is the sum of the `sum` members of all the timers with the exception of TOTAL.

- `static void close_file_sts (const stats *stp)`
  This closes the output file for the stats structure, except if the file is `stdout`.

The following function is written using the above library of functions. It also makes use of the associated global variable `all_stats[]`.

- `static void print_all_stats (void)`

  This collects and prints the statistics from all the processes at the root process. It creates a derived `MPI_Datatype` for the `stats` type, which requires a datatype for the `timer` type as well, to gather the statistics instance at the root process. Requires careful handling of the `TOTAL` timer because periodic saves mean that this timer is not stopped when printing the data.

6. CARPET Parser Implementation

CARPET programs are translated into C programs that define the global parameters and transition function of the CA. This translator, usually referred to as the *parser*, is composed of a tokeniser and a parser generated using the UNIX tools `flex` and `yacc` (or `bison`). Note that the use of the standard UNIX `lex` tool results in a tokeniser which handles comments incorrectly.

This parser is derived from the one used in the CAMEL software developed in the CABOTO project [Smith 1998]. It has been enhanced with various features according to the users' requests [Telford et al. 1999].

6.1 Tokeniser

The tokeniser (`yylex.l`) reads in text from the CARPET source file. It uses eight Left Context states to control the way it interprets text:

- `<ZERO>`

  Default start state. Reverted to in body of transition function. When the keyword "`steering`" is read, the state changes to `<STEERSTATE>`.

- `<UNO>`

  After reading "`cadef`", CARPET keywords are expected and are passed to the parser as tokens. All other strings are interpreted as identifiers, integer values, or real values and are passed as tokens, with the name or value being passed by global variables. A "`threshold`" keyword changes the state to `<CINQUE>`. When the C block close symbol "`}`" is read, indicating the end of the `cadef` block, the state changes to `<TRE>`.

- `<DUE>`

  Once the string "`update`" has been read, this state handles the parameters, i.e. the following symbols are interpreted as two comma-separated C expressions enclosed by "`(`" and "`)`". The state is then reset to `<ZERO>`.

- `<TRE>`

  All symbols are ignored and passed through to the output file except:

  - the state is changed back to `<ZERO>` after the C open block symbol "{" is read; token YSTARTCODE is generated;

  - "struct", "enum", "union" or "=" change the state to `<QUATTRO>`.

- `<QUATTRO>`

  Reads array initialisers, enum, struct and union declarations between the cadef block and the transition function body (the transition function's local declarations). Returns to `<TRE>` upon reading a ";".

- `<CINQUE>`

  Reads parameter of "threshold" directive in cadef block, similarly to state `<DUE>`. Reverts to state `<UNO>` on reading a ";".

- `<STEERSTATE>`

  Reads the name of the defined region and changes to state `<REDARG>`.

- `<REDARG>`

  Reads the limits of the defined region.

In addition, two exclusive start states are used to handle comments: `<COMMENT>` for C-style "/* ... */" comments and `<COMMENT2>` for C++-style "// ..." comments.

6.2 Parser

The parser (`yyparser.y`) is implemented as a combination of a grammar with embedded C code.

After the `cadef` block is parsed, the functions `cadef_check()`, `cadef_h_code()`, `cadef_c_code()` and `cadef_desc()` are called. Respectively, these check for missing or inconsistent `cadef` declarations, generate the header file, generate the transition function code in the output C file and fill in the `CptCADef` state table returned by the parser.

At the start of the transition function body (marked by the token `YSTARTCODE`), `neigh_code()` is called to generate the symbolic neighbourhood mapping code. The parser also transforms the `update()` statements in the transition function body into the appropriate C code.

After the update function, the steering code is generated by means of the function `cpt_steering_code()`.

6.2.1 Interface to macrocell.c

As mentioned in section 5.1.3, there is a number of functions defining the interface of the parser-generated model to `macrocell.c`. Their description follows.

- `extern void cpt_hook_init (void);`
  It defines the variables `cpt_dimx`, `cpt_dimy` and `cpt_dimz` which are equal to `CPT_X`, `CPT_Y` and `CPT_Z` respectively. More importantly, in the case of uneven decomposition, it defines the array of neighbours, `cpt_N[]`; in the even decomposition case this array is defined on declaration, but this is not possible in the case of uneven decomposition because its initialisation values depend on variables defined at run-time.

- `extern void cpt_func (CptCell *, CptCell *);`
  This defines the update function code. Its first argument is a pointer to the read copy of the cell to be updated, whereas the second argument is a pointer to the cell to be updated in the write copy of the model.

- `extern void cpt_set_state (CptCell *, int, void *, int);`
  This function is used to set the values of a given substate on contiguous cells to speci-fied values. The starting cell is pointed to by the first argument of the function, the state id is the second argument, the `void *` pointer contains the new data and the last argument defines the number of contiguous cells this operation affects.

- `extern void cpt_get_state (CptCell *, int, void *, int);`
  Similarly to `cpt_set_state()`, this function returns the substate values of a set of contiguous cells to the pointer defined in the third argument. Note that this pointer must be suitably initialised by the caller function.

- `extern void cpt_mpi_type_cell (MPI_Datatype *);`
  This function defines the derived datatype corresponding to a cell, which is in turn used to define the boundary vector datatype. The obvious way to define the cell type is by treating it as a struct, thus employing `MPI_Type_struct`. This approach is correct and guarantees that the datatype is defined correctly even when the underlying architecture consists of CPUs with various datatype representations. Portability comes at a price though. When memory for a C `struct` is allocated, there is a possibility that holes are introduced between consecutive fields. This is reflected in the derived datatype and causes MPI to call internal functions more times than it would in order to communicate these derived datatypes. To avoid this performance deterioration, the user can define the `HOMOGENEOUS` C pre-processor macro, which defines the derived datatype as an appropriately sized contiguous block of memory. This definition is *not* the default, be-cause it is not portable; it assumes that the underlying architecture is homogeneous.

- `extern void cpt_hook_finalize (void);`
  Reserved function to be executed when exiting the program. It does nothing at the mo-ment.

- `extern int cpt_thresh (CptCell *);`
  It returns the threshold condition defined by the user (see sections 5.6.3.1 and 3.3.24 for more on the threshold condition).

6.2.2 Steering Code Generation

6.2.2.1 Steering Related Types

The following structure types are defined in `yyparser.y`:

6.2.2.1.1 **CptRegion**

```
typedef struct _CptRegion {
    char    *name;
    int bounds[6];
    struct _CptRegion *next;
} CptRegion;
```

This defines a single-link list containing the data for each of the defined regions. The associated functions are as follows:

- `static CptRegion *cpt_make_region (char *name, int bounds[6],`
  `                                   CptRegion *list)`
  This function adds a region to the list pointed by `list`, having the members pointed by the first two arguments of the function. It returns a pointer to the head of the list.

- `static void cpt_free_region (CptRegion *list)`
  This frees the dynamically allocated memory of the contents of the list pointed by `list`.

- `static CptRegion *cpt_find_region (char *name, CptRegion *list)`
  This makes a search in `list` for a region with the same name member as `name`. It returns a pointer to such a `CptRegion` if found, or `NULL` otherwise.

- `static void cpt_check_region (CptRegion *list, int dim)`
  This checks the regions in `list` to verify that if the model is 1-dimensional or 2-dimensional. The bounds of the regions are not specified for unused dimensions.

6.2.2.1.2 `CptReduction`

```
typedef struct _CptReduction {
  char name[BUFSIZ];
  int type;
  int unsign;
  char *neutral;
  struct _CptReduction *next;
} CptReduction;
```

This is a single-link unordered list, with descriptions of the reduction operations in the CARPET program. The `type` member is a handle to the datatype of the arguments in the reduction operation and `unsign` is a flag whether the datatype is unsigned or not. The `neutral` member is the neutral element for the reduction. We briefly discuss the related functions.

- `static CptReduction *cpt_make_reduction (char *name, int type,`
  `int unsign, CptReduction *list)`
  This function checks the reduction operations already in `list` for a reduction with the same characteristics as the one to be inserted. All the details of the new reduction are available from the argument list of the function with the exception of the neutral element, provided by the `get_neutral()` function discussed below. Returns a pointer to the head of the list.

- `static void free_reduction_list(CptReduction *list)`
  This frees all the elements in `list`.

- `static void emit_red_func (CptReduction *p)`
  This routine outputs the reduction function corresponding to the reduction pointed at by `p` to the generated C file. Information about the prototype and how to write a reduction function is available from section 3.3.19.

- `static char *get_neutral (CptReduction *p)`
  This function returns the neutral element for the reduction pointed at by `p`. The neutral element depends on the type of data and the reduction operation. If the combination of the two above members is not matched in the function code, `NULL` is returned. In this

case the user supplies the neutral element for the operation implicitly in the reduction function that they provide. See section 3.3.19 for the importance of the neutral element.

- `static int print_red_func (CptReduction *p)`
  This function prints only one line in the generated C file, containing the operator corresponding to the reduction pointed by `p`. For example, if the function is `max`, it prints the following:

  ```
  res = MAX (res, tmp_data[i]);
  ```

  The definitions of the macros `MIN` and `MAX` are emitted in the generated file when the `cpt_write_reductions()` function is called. The function returns -1 if the operation is unknown, or 1 in the normal case.

- `static void print_all_reduce (CptReduction *p)`
  This function also prints only one line in the generated C file. It outputs the MPI global reduction statement.

6.2.2.2 Steering Related Global Variables

The following global variables in `yyparser.y` are related to the steering code generation:

- `static CptRegion *cpt_region_list=NULL;`
  The list of all the regions, initialised by `cpt_make_region()`.

- `static CptRegion *current_region;`
  A pointer, used when outputting the steering function.

- `static CptReduction *cpt_reduction_list=NULL;`
  The list of all the reductions, initialised by `cpt_make_reduction()`.

- `static char redop_name[BUFSIZ];`
  This is used to store the reduction operation name when outputting the reduction functions.

6.2.2.3 Steering Related Functions

The functions in `yyparser.y` associated with the generation of the steering code are as follows:

- `static void cpt_steering_code (void)`
  This function outputs the `cpt_abort()` and `cpt_set_param()` functions in the generated C file.

- `static void cpt_write_reductions (void)`
  This function writes the following to the generated C file:
  - a list of steering-related standard header files which should be included;
  - a list of definitions used internally, such as `MIN` and `MAX`;
  - the reduction functions, by calling `emit_red_func()` for all the members of `cpt_reduction_list`.

  It also writes the prototypes of the generated reduction functions to the generated header file and frees the list by means of `free_reduction_list()`.

6.3 Parser library interface

The parser is built as a library (`libcpt_parse`) with the following interface (declared in `cpt_parse.h`):

- ```
  int cpt_init (const char *carpet_file, const char *c_file,
                const char *h_file,
                void (*error_handler)(int code, int line))
  ```
  This function is called to start the parsing process. It opens the pathname `carpet_file` as the input CARPET source file, `c_file` as the output C file, and `h_file` as the output C header file. The tokeniser and parser are initialised.

  `error_handler()` is a pointer to user-supplied callback function which is called when a parser error occurs. The parameters passed to `error_handler()` are the error code and CARPET source file line number respectively. If `line` is 0, the error is of a *global* nature (i.e. failed to open file); codes 900 and above are considered *warnings* and do not prevent the output files from being generated.

Returns -1 on error or 0 otherwise.

- `int cpt_parse (CptCADef *cadef)`
  This function is called to perform the parsing process. The output files specified in the `cpt_init()` call are written to and the state table pointed to by `cadef` is filled in with information about the CARPET program. Note that the `cadef->st` and `cadef->pt` tables are allocated by the parser and must be freed by the user when no longer required.

  Returns 0 on success or number of errors found.

- `int cpt_finalize (void)`
  This function closes the three files given in the call to `cpt_init()`.

  Always returns 0.

- `void cpt_error_message (int code, char *message, int length)`
  This function copies up to `length` characters of an error message corresponding to error code `code` into the user-supplied buffer pointed to by `message`. Error messages are defined by the parser library.

7.   GUI–CA Engine Communication

This section presents the design of the protocol implemented for the communication between the GUI and the CA Engine of CAMELot. The implementation of the corresponding functions is also discussed in detail.

7.1 General Remarks

7.1.1 Communication Abstraction

It was decided to implement the protocol using BSD sockets. The reason for choosing sockets is that they provide a simple programming interface. This interface is more advanced and better documented than those of pipes or FIFOs. MPI-1 could not be used because its specification does not allow processes to start at different times, which is essential for the application since the GUI spawns the macrocell processes. The reason for choosing Berkeley sockets instead of TLI is that they have been established over the past years and are widely supported across platforms [Stevens 1990].

Our protocol was implemented over TCP, which provides a bi-directional, connection oriented channel of communication. The connection is established at the beginning of the run and is not terminated until a request to exit the program is received (`EXITCODE`).

The functions implemented are not socket dependent. Their prototypes do not contain the sockets as arguments and are thus easily modifiable.

7.1.2 Socket Instances

There are two socket instances in each of the GUI and CA Engine, one for visualisation and one for the other protocol requests, named `vis_sockfd` and `prot_sockfd` respectively. In both cases, the GUI acts as a server, which is expected since the GUI process spawns the CA Engine.

On the GUI side, the program calls `socket`, `bind` (with `sin_port` set to 0 so as to have the system assign the port number), `listen`, then spawns the CA processes and calls `accept` twice on the initial socket to get `prot_sockfd` and `vis_sockfd` respectively. On the CA Engine side, the program calls `socket` and `connect` twice, in order to initialise `prot_sockfd` and `vis_sockfd`. This is done by calling the function `start_client()` twice.

The name of the host and the name of the port are passed to the master macrocell process through the -H and -P command line arguments of the macrocell program, respectively.

7.1.3 Header Format

In order for the two sides to exchange messages, the communication initiator must send a valid `req_code` as defined in the header file `constants.h` and shown in Table 9. Of those, `FINISHED` and `BATCH` are reserved for internal use in `macrocell.c` and `IGNORED` is used as an acknowledgement only. `VIS_PACK` is used by the CA Engine to communicate visualisation data, and `GEN_NO` to send generation numbers. `OVER_W` is only transmitted from the CA Engine in the special case described in the discussion of `serv_periodic_save()` in section 7.5.7. All other codes are used only by the GUI. The receiving side ignores messages which do not have a valid `req_code`.

7.1.4 Spatial Entities

The co-ordinates of a spatial entity (plane, line or cell) are uniformly passed to the function by means of the integer array, `pos[3]`. The exception to this rule is function `req_add_plane()`, which encapsulates the array to its `plane` argument, as explained in section 8.4.3. This convention implies that an entity will always extend to its maximum dimensions, thus leaving sub-entity display for the GUI. Since valid co-ordinates for each dimension range from 1 to the maximum number of cells in the axis, in order to "free" a dimension the appropriate element of the position array has to be set to 0.

7.2 Auxiliary Functions

7.2.1 Socket Functions

The following are the socket-related functions of the CAMELot software. They are implemented in file `sock.c`, and their prototypes can be found in `common.h`.

- `int readn (int fd, char *ptr, int nbytes)`
  Reads `nbytes` bytes from file descriptor `fd` into the supplied buffer `ptr`. It assumes that the file descriptor has been opened and the pointer is appropriately initialised to hold the data. This function is a wrapper for `read(2)`. It returns the number of bytes actually read.

```
typedef enum {IGNORED = INT_MIN,
              EXITCODE = -13,
              FINISHED = 1,
              OVER_W = 333,
              SAVE_REQUEST = 1111,
              SET_FOLD,
              SET_LOAD,
              VIEW_STATE,
              SET_STATE,
              SET_PARAM,
              GET_PARAM,
              EVOLVE,
              LOOP,
              RESUME,
              TERMINATE,
              PAUSE,
              ADD_PLANE,
              DEL_PLANE,
              READ_PROJECT,
              SAVE_PROJECT,
              PERIODIC_SAVE,
              VIS_PACK,
              GEN_NO,
              SET_MINMAX,
              BATCH} req_code;
```

Table 9: Enumerated type `req_code`

- int writen (int fd, char *ptr, int nbytes)
  Writes nbytes bytes to file descriptor fd from the supplied buffer ptr. It assumes that
  the file descriptor has been opened. This function is a wrapper for write(2). It returns
  the number of bytes written.

- int start_client (u_short port, char *hostname)
  This function initialises a client by connecting to the process running on port port on
  host hostname. It assumes that a TCP connection must be made and takes the address

of the host using the `gethostbyname(3N)` function. In the normal case it returns the socket descriptor returned by `socket(5)`, after achieving a connection using `connect(3N)`. In the case of our application, the client is the CA Engine. The function returns a negative integer if any of the calls fails.

7.2.2 Acknowledgements

Depending on the function executed, the CA Engine should return an acknowledgement to the GUI, regarding the success of the requested action.

Acknowledgements are handled by the following two functions.

- `static int send_ack (req_code ack)`
  This function, local to `macrocell.c`, transmits the `req_code` that initiated the action as an acknowledgement for a successfully executed task, or a negative error code if the caller function failed. The function returns `ack`, except if `writen()` fails, in which case it returns a negative value. The special negative `req_code`, `IGNORED` might also be transmitted and thus returned. This does not indicate a failure of the function.

  Acknowledgements are handled in the GUI-side by the `get_ack()` function, which compares the received acknowledgement code with the one expected in each case.

- `int get_ack (req_code request)`
  The function is implemented in file `guicomms.c`; its prototype is listed in `guicomms.h`. It returns:

  - `request`, if this is the value of the message read;
  - 0, if the message read is `IGNORED`;
  - a negative integer, otherwise.

7.3 Requests

Here we describe protocols and the implementations of the functions on each side, with respect to each of the values of `req_code`.

SAVE_REQUEST

The GUI requests that the values for a certain substate be written to a file whose name is transmitted. An acknowledgement is expected at the GUI side.

Sender	Token	Type
GUI	SAVE_REQUEST	req_code
GUI	substate	int
GUI	strlen (fname)	int
GUI	fname	char *
CA	SAVE_REQUEST	req_code

SET_FOLD

This is a request to set the active folds manually. An acknowledgement is expected after completion of the action.

Sender	Token	Type
GUI	SET_FOLD	req_code
GUI	start_fold	int
GUI	end_fold	int
GUI	fname	char *
CA	SET_FOLD /	req_code
	IGNORED	

SET_LOAD

The GUI requests that the specified substate values of all cells in the CA Engine be set to those listed in the specified file. An acknowledgement is expected from the CA Engine.

Sender	Token	Type
GUI	SET_LOAD	req_code
GUI	substate	int
GUI	strlen (fname)	int
GUI	fname	char *
CA	SET_LOAD	req_code

VIEW_STATE

The GUI transmits the co-ordinates of an entity and gets the data for the substate and the generation that the data were collected. This request is used by the "Edit Substate" GUI facility.

Sender	Token	Type
GUI	VIEW_STATE	req_code
GUI	pos	int[3]
GUI	substate	int
CA	cpt_generation	int
CA	data	char *
CA	VIEW_STATE	req_code

SET_STATE

The GUI transmits appropriate values and requests that the substate be set in the CA Engine. An acknowledgement finishes the communication. This request is used by the "Edit Substate" GUI facility.

Sender	Token	Type
GUI	SET_STATE	req_code
GUI	pos	int[3]
GUI	substate	int
GUI	data	char *
CA	SET_STATE	req_code

SET_PARAM

This request concerns the modification of `cadef` global CARPET parameters. More than one parameter can be set, as their number is written to the socket. The reply from the CA Engine is an acknowledgement.

Sender	Token	Type
GUI	SET_PARAM	req_code
GUI	no_of_params	int
for (i)		
GUI	param_id[i]	int
GUI	value[i]	float
end for		
CA	SET_PARAM	req_code

GET_PARAM

This request gets the value of one global CARPET parameter. There is no acknowledgement in this case.

Sender	Token	Type
GUI	GET_PARAM	req_code
GUI	param_id	int
CA	param[param_id] /	int /
	IGNORED	req_code

EVOLVE

The GUI requests the evolution of the CA Engine for a given number of generations. No reply is expected.

Sender	Token	Type
GUI	EVOLVE	req_code
GUI	num_gens	int

LOOP

This is a request for CA Engine execution until further notice. No reply is anticipated.

Sender	Token	Type
GUI	LOOP	req_code

TERMINATE

This request terminates CA Engine execution, but it does not cause the program to exit. The user can instruct a new run of the Engine. Visualisation planes are removed from the data structures and the generation is zeroed. An acknowledgement that execution has stopped is returned to the GUI through the communication channel.

Sender	Token	Type
GUI	TERMINATE	req_code
CA	TERMINATE / IGNORED	req_code

PAUSE

This requests the CA Engine to pause execution. Its difference from TERMINATE is that in this case the visualisation planes are not affected and the generation is not zeroed. When paused, the CA Engine can accept requests and can then be restarted by:

- EVOLVE or LOOP, in which case the visualisation list will be reinitialised but not emptied (unlike TERMINATE). This effects to the planes being displayed immediately;

- RESUME, in which case no changes to the lists are imposed, except those explicitly requested while the Engine was paused.

No acknowledgement that the Engine is paused is transmitted to the GUI.

Sender	Token	Type
GUI	PAUSE	req_code

ADD_PLANE

This is a request to add a visualisation plane to the CA Engine. An acknowledgement is expected at the GUI side *except* if the plane is IGNORED. The protocol for ADD_PLANE is explained in Section 8.4.

Sender	Token	Type
GUI	ADD_PLANE	req_code
GUI	pos	int[3]
GUI	substate	int
GUI	vis_step	int
CA	ID /	int /
	IGNORED	req_code
CA	ID_same /	req_code
	(NOTHING)	
CA	ADD_PLANE /	req_code
	(NOTHING)	
GUI	VIS_PACK	req_code

DEL_PLANE

The GUI requests the deletion of a plane identified by its ID. An acknowledgement is sent to the GUI.

Sender	Token	Type
GUI	DEL_PLANE	req_code
GUI	ID	int
CA	DEL_PLANE / IGNORED	req_code

READ_PROJECT

Requests that the CA Engine initialise its state from the data in the file whose name is transmitted. Communication is finished with an acknowledgement.

Sender	Token	Type
GUI	READ_PROJECT	req_code
GUI	strlen (fname)	int
GUI	fname	char[strlen(fname)]
CA	READ_PROJECT	req_code

SAVE_PROJECT

The GUI requests that the current state of the CA Engine be saved in a set of project files. The resulting files can be used for the READ_PROJECT operation, as well as SET_STATE.

Sender	Token	Type
GUI	SAVE_PROJECT	req_code
GUI	strlen (fname)	int
GUI	fname	char[strlen(fname)]
CA	SAVE_PROJECT	req_code

PERIODIC_SAVE

The GUI initiates periodic saving of project data.

Sender	Token	Type
GUI	PERIODIC_SAVE	req_code
GUI	strlen(fname)	int
GUI	fname	char[strlen(fname)]
GUI	save_step	int
CA	OVER_W / 0	req_code
CA	PERIODIC_SAVE	req_code

SET_MINMAX

The GUI sets the minimum and maximum values for the colour mapping of a substate.

Sender	Token	Type
GUI	SET_MINMAX	req_code
GUI	substate	int
GUI	l_minmax	double[2]
CA	SET_MINMAX / IGNORED	req_code

IGNORED

According to the state of the Engine, the following requests are IGNORED:

- TERMINATE and PAUSE, if the Engine is Stopped;
- PAUSE, if the Engine is Paused;
- EVOLVE and LOOP, if the Engine is Running;
- RESUME, if the Engine is Running or Stopped;
- FINISHED, if the Engine is Paused or Stopped.

Sender	Token	Type
CA	IGNORE	req_code

Other reasons for the CA Engine to transmit IGNORED are as follows:

- if the plane to be added is already in the visualisation list;
- if the plane to be deleted does not exist;
- if the transmitted spatial entity is invalid;
- if the parameter id for the parameter to be transmitted is invalid;
- if the suggested minimum or maximum values for the colour mapping of a substate are inadequate.

If in any of these cases an acknowledgement is expected, `IGNORED` is transmitted to the GUI.

EXITCODE

This request causes the CA Engine program to exit.

Sender	*Token*	*Type*
GUI	EXITCODE	req_code

FINISHED

This is not available on the GUI side; it is used in the CA Engine after an `EVOLVE` request has been completed so as to reset internal variables.

Sender	*Token*	*Type*
Not Transmitted		

RESUME

This is one of three ways to restart the CA Engine after it has been Paused. Explained under `PAUSE`.

Sender	*Token*	*Type*
GUI	RESUME	req_code

VIS_PACK

This is a visualisation packet identifier, sent as a header from the CA Engine to the GUI before sending the visualisation data. This is one of the two protocol functions performed

through `vis_sockfd`, the other one being `GEN_NO`. These two are also the only ones to be initiated by the CA Engine.

Sender	Token	Type
CA	VIS_PACK	req_code
CA	ID	int
CA	valsize	int
CA	minmax[2]	double[]
CA	value[valsize]	char[]

`GEN_NO`

This is a generation number identifier, sent from the CA Engine to the GUI after each generation has been executed. It is followed by the current generation number.

Sender	Token	Type
CA	GEN_NO	req_code
CA	cpt_generation	int

`OVER_W`

This is a special kind of acknowledgement sent by the CA Engine when files could be overwritten as a result of the periodic save. See section 7.5.7 for more details.

`BATCH`

This is not available on the GUI side either. It is used in the CA Engine instead of `EVOLVE` when the program is run in standalone mode so that `rv()` initialises the status of the CA Engine and exchanges boundaries.

Sender	Token	Type
	Not Transmitted	

7.4 Implementation of GUI Functions

The following are the protocol-related functions contained in the `libcmtguicomms` library linked with the GUI. They are all implemented in file `guicomms.c`; their prototypes are listed in `guicomms.h`. The function arguments for functions prefixed `req_` are to be transmitted to the other side, except if otherwise stated. All functions return the `req_code` if finished successfully. Functions receiving an `IGNORED` acknowledgement return 0. We discuss them with respect to their context.

7.4.1 Substate related

- `int req_save_request (int substate, char *filename)`
  The GUI requests that the `substate` values for all cells in the CA Engine be written to file `filename`. The function merely implements the protocol.

- `int req_set_load (int substate, char *filename)`
  Set the `substate` values for all cells in the Engine to those listed in file `filename`.

- `int req_view_state (int pos[3], int substate,`
  `                     CptStateType st_type, int *gen, void *value)`
  The `pos[]` array contains the co-ordinates of the cell, line or plane whose `substate` data are to be retrieved. The results are returned in the `value` array, which has to be initialised by the caller function. The generation number of the state is returned in the `gen` pointer; no memory allocated for it either. Only `pos` and `substate` are written to the CA side. `st_type` is used to calculate the total size of the `value` argument and accordingly receive data.

- `int req_set_state (int pos[3], int substate,`
  `                    CptStateType st_type, void *value)`
  This requests that the substate of the entity in `pos[]` be set to `value`. Symmetric to `req_view_state`, but it does not affect the generation of the CA Engine.

- `int req_set_param (int no_of_params, int *param_id,`
  `                    float *value)`
  This function is concerned with the modification of `cadef` global CARPET parameters. Their value can only be of type `float`. If `no_of_params` parameters are to be set, their ID is stored in the `param_id` array and the corresponding values can be found

in `value`. The size of both arrays is `no_of_params`. After sending `no_of_params`, the function loops over an index `no_of_params` times and writes the respective values of `param_id` and value to the socket.

- `int req_get_param (int param_id, float *value_ptr)`
  This function reads the parameter indexed `param_id` from the CA Engine. The value of the parameter read is stored in `value_ptr`. If the value read is `IGNORED` it returns 0, otherwise it returns `GET_PARAM`. The caller function must allocate memory for `value_ptr`.

7.4.2 Program Flow Management

The `PAUSE`, `LOOP`, `TERMINATE`, `EXITCODE` and `RESUME` `req_codes` are implemented by issuing a simple `writen` call; no special function has been developed for them. `FINISHED` is not available to the GUI.

- `int req_evolve (int num_gens)`
  The GUI requests the CA Engine evolution for `num_gens` generations. The implementation is trivial.

- `int req_set_fold (int start_fold, int end_fold)`
  This is a function to set the starting and finishing active folds of the CA manually. The implementation is straightforward.

- `int req_set_minmax (int substate, double min, double max)`
  This function sets the minimum and maximum values for the given substate so as to be used on the CA side for the colour-mapping. It implements the protocol.

7.4.3 Visualisation Functions

- `int req_add_plane (plane *pl_ptr, int *ID_same)`
  This is a request to add a visualisation plane to the CA Engine. The `plane` definition as well as the discussion of the function are deferred to section 8.4.

- `int req_del_plane (int plane_id)`
  The GUI requests the deletion of the plane numbered `plane_id`. The function implementation is detailed in section 8.5.

7.4.4 Configuration (Project) Related

- `int req_proj_read (char *filename)`
  The function only transmits the length of `filename` followed by the filename itself
  and then blocks for the acknowledgement. `filename` is used as a root for the files to
  be read. The filename construction as well as the CA Engine actions are detailed in the
  discussion of `serv_proj_read`.

- `int req_proj_save (char *filename)`
  Similar to `req_proj_read`.

- `int req_periodic_save (char *filename, int period, int *cfs)`
  Requests the CA Engine to save the state of the system periodically in files whose
  name has the given root `filename`. The argument is an integer passed *by-reference*
  conveying the result of `check_fs()` on the macrocell side (see section 7.5.7 for more
  details). The implementation of this function is trivial.

7.4.5 Other functions

The rest of the functions in file `guicomms.h` are as follows:

- `int consume_vis_pack (void)`
- `void GUI_check_pos (int *)`
- `int GUI_get_val_size (const int *)`
- `int get_max_size (const unsigned int *)`

These functions are discussed in section 8.3.

7.5 Implementation of the CA Engine Functions

7.5.1 General Remarks

The following functions are called to serve the corresponding GUI side requests. These
functions are asymmetric to their GUI-side counterparts, in that they are invoked immedi-
ately after the request has been received and read in the necessary data internally from the
communication channel. Thus they have a void argument list. They return the `req_code`
that initiated them if successful, `IGNORE` if they did not perform a change for the reason
explained previously or a negative error code in other cases. The acknowledgement, where
applicable, is sent by their caller function, `rv()` (see section 7.5.2).

The functions receiving a filename first read its length through the socket. In order to use the filename as a character pointer we `NULL` terminate it. So, memory for an extra character must be allocated.

Data which need to be known to all the processes are broadcast to them using the `MPI_Bcast` function. Data are scattered or gathered from or to the root process using `MPI_Scatter` and `MPI_Gather` respectively. The root process in all collective communications is process 0.

7.5.2 *Function* `rv()`

The function responsible for request handling on the CA Engine side is

```
static int rv (req_code request)
```

For each `req_code`, with the exceptions of `EVOLVE`, `BATCH`, `LOOP`, `RESUME`, `PAUSE`, `FINISHED`, `GEN_NO`, `OVER_W` and `EXITCODE` there is a function on the CA Engine side to handle the request. The function `rv()` consists of a `switch` statement each case of which calls the appropriate function and then transmits the acknowledgement to the GUI (where applicable).

The reason why there are specific requests which do not have corresponding functions is that they only affect the state of the CA Engine and do not require significant computation or process interaction. The state of the CA Engine is maintained within `rv()` with the use of two local static variables, `paused` and `started`. In addition to these, another static variable, `init_gen`, denotes whether `cpt_generation` has been explicitly set by any of the initialisation functions (e.g. `serv_proj_read()` discussed in section 7.5.7) and should thus be preserved. We will describe the implementation of the handling mechanism for each of these requests. `OVER_W` is omitted as it is only used as an acknowledgement for function `serv_periodic_save()`, see section 7.5.7. `GEN_NO` is not discussed here either as it is only transmitted by the CA Engine, see section 7.3.

`EVOLVE`
This is `IGNORED` if already `started` and not `paused`. Otherwise, the number of generations to be run is read and broadcast to all the processes in the global variable `num_gens`. When this is received:

- function `set_gen()` is called to reset the visualisation generation in all the planes in the list (see section 8.3.5.2);
- `init_boundaries()` is called;
- `started` is set;
- `paused` is cleared;
- `init_gen` is cleared.

Returns EVOLVE or IGNORED as discussed above.

BATCH

The request is IGNORED if `started` or `paused` or if `prot_sockfd` is set. The number of generations is passed to `macrocell.c` by means of the `-n` argument. Similarly to EVOLVE:

- `set_gen()` is called;
- `init_boundaries()` is called;
- `started` is set;
- `paused` is cleared;
- `init_gen` is cleared.

BATCH or IGNORED may be returned as usual.

LOOP

The same as EVOLVE, only that there is no number of generations to be read, as this requests an infinite loop.

RESUME

This is IGNORED if `paused` is not set. It simply clears `paused` and `init_gen`.

PAUSE

This is IGNORED if the Engine has not `started` or if it is already `paused`. It sets `paused` and returns PAUSED.

FINISHED

This pseudo-`req_code` is used by `run()` to clear the `started` variable. It is IGNORED if the Engine is not `started`.

EXITCODE

Just returns `EXITCODE`.

7.5.3 File and Socket I/O

7.5.3.1 Data Handling

Data communicated to the GUI or saved in a file follow the rule that the *x* dimension changes fastest, followed by *y*, followed by *z*. In other words, if the data in question are of size `xextent*yextent*zextent`, the CA Engine will write them looping `zextent` times over `yextent`, sending `xextent` data each time. Technically, these loops are not executed when writing, but the effect to the order of the written data is the one described above. `xextent`, `yextent` and `zextent` are determined by means of an array, `pos[3]`. If a co-ordinate of this array is set to 0, then the corresponding extent may be `DIMX/DIMY/DIMZ`, except if the dimension is not used, in which case the extent equals 1. If the co-ordinate is greater than 0, then the corresponding extent is equal to 1. This assumes that the GUI enumerates the cells in each dimension starting from 1, contrary to the CA Engine which enumerates from 0.

7.5.3.2 Writing Data

Functions that write to a file or socket contain the temporary data storage variable `tmp_data` of type `unsigned char*`. This is used to get the data from the CA Engine part belonging to each process and gather them in the root process. In the general case its size is `CPT_DIMX`[12]`*DIMY*DIMZ*cpt_state_size[stateid]` (the number of elements in each process multiplied by the size of each element of a given substate). On the other hand, the root process requires an extra variable `tmp_data2` of the same datatype and adequate size (generally `DIMX*DIMY*DIMZ*cpt_state_size[stateid]`) in which to collect the data. All the arrays above contain actual data, stripped of boundary data. This is achieved by means of function `get_x_line()` which skips the boundaries when traversing the model.

The approach in gathering the data to the root process is very different depending on whether even decomposition is assumed or not. We will describe these cases separately.

---

[12] `CPT_DIMX[cmtWorldRank]` in the case of uneven decomposition

7.5.3.2.1 Even Decomposition Data Collection

The data are gathered into the root process using `MPI_Gather`. The data in `tmp_data` are contiguous *per process*. It would be an error to gather them into the root process using a simple `MPI_Gather` call. This would mean that the first `NPROCS` *x*-lines of process 0 would be considered as one *x*-line of the model. In order to interleave the process lines while gathering, we create derived `MPI_Datatypes`, both for sending and for receiving data. The process of creating these datatypes resembles the rationale of the data decomposition to folds consisting of strips, as discussed in section 5.4. For the sending datatype, we create first a vector datatype, `send_strip_vec`, packing `DIMY*DIMZ` blocks of `CPT_F_X` contiguous elements. The distance between the first elements of two consecutive contiguous blocks is set to `CPT_DIMX` elements. This is the MPI way to represent a strip on the sender. We then create another datatype, `send_strip_UB_type`, using the `MPI_Type_struct` call, to fix the extent of this datatype to `CPT_F_X`. This is a technical requirement for MPI and it is achieved by setting the upper bound of the datatype to an address `CPT_F_X` elements away from its beginning.

The receiver end creates a strip vector, `recv_strip_vec`, similarly to the sender, only that the stride between the first elements of two consecutive contiguous blocks is set to `DIMX`, i.e. the *x*-size of the receiving buffer. This is the building block of the `recv_fold_type` derived datatype, naturally consisting of `NFOLDS` strips. Finally, we create a fixed extent datatype `recv_fold_UB_type`, in the same manner as above and with the same extent. `MPI_Gather` is called so that `NFOLDS` elements of type `send_strip_UB_type` are sent from each process and process 0 receives one element of type `recv_fold_UB_type` from each process. The implementation of this procedure is shown in Table 10.

```
/* Type strip */
    MPI_Type_vector (DIMY*DIMZ, CPT_F_X, CPT_DIMX,
                     cpt_state_mpidt[stateid],
                     &send_strip_vec);
    MPI_Type_commit (&send_strip_vec);

/* Type strip with fixed extent for gather */
    types[0] = send_strip_vec;
    types[1] = MPI_UB;

    displacements[0] = 0;
    MPI_Address (&(tmp_data[0]), &start_address);
    MPI_Address (&(tmp_data[CPT_F_X*cpt_state_size[stateid]]), &address);
    displacements[1] = address-start_address;

    block_lengths[0] = 1;
    block_lengths[1] = 1;

    MPI_Type_struct (2, block_lengths, displacements, types,
```

```
                                &send_strip_UB_type);
    MPI_Type_commit (&send_strip_UB_type);

/* Type strip. Different than send_ in stride since buffer is bigger */
    MPI_Type_vector (DIMY*DIMZ, CPT_F_X, DIMX, cpt_state_mpidt[stateid],
                       &recv_strip_vec);
    MPI_Type_commit (&recv_strip_vec);

/* Type fold */
    for (i = 0; i < NFOLDS; i++) {
      types[i] = recv_strip_vec;
    }

    displacements[0] = 0;
    MPI_Address (&(tmp_data2[0]), &start_address);

    for (i = 1; i < NFOLDS; i++) {

      MPI_Address(&(tmp_data2[i*CPT_NPROCS*CPT_F_X*cpt_state_size[stateid
]]),
      &address);
      displacements[i] = address-start_address;
    }

    for (i = 0; i < NFOLDS; i++) {
      block_lengths[i] = 1;
    }

    MPI_Type_struct (NFOLDS, block_lengths, displacements, types,
                       &recv_fold_type);
    MPI_Type_commit (&recv_fold_type);

/* Type fold with fixed extent for gather */
    types[0] = recv_fold_type;
    types[1] = MPI_UB;

    displacements[0] = 0;
    MPI_Address (&(tmp_data2[0]), &start_address);
    MPI_Address (&(tmp_data2[CPT_F_X*cpt_state_size[stateid]]),
                 &address);
    displacements[1] = address-start_address;

    block_lengths[0] = 1;
    block_lengths[1] = 1;

    MPI_Type_struct (2, block_lengths, displacements, types,
                       &recv_fold_UB_type);
    MPI_Type_commit (&recv_fold_UB_type);

    MPI_Gather (tmp_data, NFOLDS, send_strip_UB_type,
                 tmp_data2, 1, recv_fold_UB_type, 0, cmtCommCommand);
```

Table 10: The code for the derived datatype used for interleaved gathering of data in the root process. Taken from function `cmt_write()`.

It should be noted that this non-trivial and costly procedure introduced in release 1.2 of CAMELot eliminates the need for the root process to rearrange the data from folds to normal line representation. However, there are functions, namely `tx_vis_pack()` and

`serv_view_state()`, which do not employ this method. The reason is that these functions may need to handle a subset of the substate data, which makes the implementation of the strategy more complicated.

In order for the aforementioned functions to interleave the process lines while gathering, we create a derived `MPI_Datatype` called `recv_vec`. This in turn contains another derived datatype called `send_vec`. The latter is created using the `MPI_Type_vector` command and it generally contains `DIMY*DIMZ` blocks of elements of a specific datatype, each having length equal to `CPT_DIMX` with stride `DIMX`. In other words, this is a vector which leaves enough space for the whole *x*-line of the model (`DIMX`), yet carries the data of one process (`CPT_DIMX`). The `recv_vec` is then created using the `MPI_Type_struct` function, as a two-element struct, the former being `send_vec` and the latter the pseudo `MPI_Datatype` `MPI_UB`. The displacement for the upper bound is set to `CPT_DIMX*state_size`, i.e. enough for the data of one process. The arguments of the `MPI_Gather` call are set in such a way, so that the processes send contiguous data which are rearranged in the receiver process, as shown in Table 11. It should be noted that these structures are local to each function. A global datatype variable cannot be constructed, since this depends on the datatype of the data to be transferred. This statement is true for the other method of data transmission as well.

```
MPI_Type_vector (my_y*my_z, my_x, my_x*work_size,
                 cpt_state_mpidt[substate], &send_vec);
MPI_Type_commit (&send_vec);

displ[0] = 0;
displ[1] = my_x;
blocklengths[0] = 1;
blocklengths[1] = 1;
types[0] = send_vec;
types[1] = MPI_UB;
MPI_Type_struct (2, blocklengths, displ, types, &recv_vec);
MPI_Type_commit (&recv_vec);

MPI_Gather (tmp_data, my_x*my_y*my_z, cpt_state_mpidt[substate],
            tmp_data2, 1, recv_vec, 0, cmtCommWork);
```

Table 11: The code for the derived datatype used for interleaved gathering of data in the root process. Taken from function `serv_view_state()`. This code implies the need to rearrange the data from fold to normal representation before writing them.

The data collected using the above process are fragmented across the *x*-axis in strip sized portions because of the folded block-cyclic decomposition and need to be rearranged. We

use the `tmp_data3` array of size equal to that of `tmp_data2`, as an argument to function `get_write_ptr()` which returns data ready for transmission. It should be noted that `get_write_ptr()` is only executed in the root process. The size of the temporary data storage variables is of particular importance and is further discussed in section 7.5.3.6. These data structures are allocated memory in every call of the functions according to the requirements and are freed before exiting the function.

7.5.3.2.2 Uneven Decomposition Data Collection

In this case, point-to-point sends and receives are used to communicate the data. The reason is that strips have different sizes and `MPI_Gather` and `MPI_Gatherv` are not flexible enough to handle interleaving variable lengths of data from multiple processes.

For each process role (send-receive) two types of vectors are needed, a small and a large one. Recall from the discussion of function `calc_x_sizes()` in section 5.4.1 that we decided to distribute extraneous cells to the processes from last to first, and place them in strips from first to last. As far as the senders are concerned, the *x*-size of the large strip can be found in `CPT_F_X[cmtWorldRank][0]`, whereas the small size will be found in `CPT_F_X[cmtWorldRank][first_small_strip_ind[cmtWorldRank]]`. The process may not have different sized strips, and this is easily tested by comparing `first_small_strip_ind[cmtWorldRank]]` against its default value, `NFOLDS`. Note that this is not a valid index for the `first_small_strip_ind` array. Similarly with the above, for the receiver the *x*-size of the large one can be found in `CPT_F_X[last][0]`, and the smallest sized strip, is in `CPT_F_X[0][first_small_strip_ind[0]]`.

All four vector types consist of `DIMY*DIMZ` blocks of contiguous data, with sizes as above. Similarly to the even decomposition case, send vectors differ from receive vectors in the *stride*, i.e. the distance between the start of two consecutive contiguous blocks. Send vectors have a stride equal to the *x*-dimension of the process, `CPT_DIMX[cmtWorldRank]`, whereas receive vectors have a stride equal to the *x*-dimension of the model, `DIMX`.

Data exchange is achieved with immediate receives issued from the root process and standard sends issued from each process (including the root process for ease of implementation). The root process issues `NFOLDS*CPT_NPROCS` receives, and each process issues `NFOLDS` sends, one for each strip, with the appropriate sizes. The tags are defined as a sequence starting with 0 and incrementing by 1 for each strip encountered when traversing the original model (e.g. the second tag equals to 1 and corresponds to strip 0 of process 1).

A summary of the code used for the case of uneven decomposition is shown in Table 12. An interesting technical issue has to do with the traversal of the data received on process 0. This is accomplished with two nested for loops across strips and then across processes. It should be noted that the order of these loops should not be swapped, otherwise the index, calculated incrementally, in the receiving array will be miscalculated

```
/* Send types */
    MPI_Type_vector (DIMY*DIMZ,CPT_F_X[cmtWorldRank][0],
                     CPT_DIMX[cmtWorldRank], cpt_state_mpidt[stateid],
                     &send_strip_vec_large);
    MPI_Type_commit (&send_strip_vec_large);
    if (NFOLDS != first_small_strip_ind[cmtWorldRank]) {
      MPI_Type_vector (DIMY*DIMZ,
              CPT_F_X[cmtWorldRank][first_small_strip_ind[cmtWorldRank]],
              CPT_DIMX[cmtWorldRank], cpt_state_mpidt[stateid],
              &send_strip_vec_small);
      MPI_Type_commit (&send_strip_vec_small);
    }               /* End if (first_small_strip_ind[cmtWorldRank]) */

/* Receive types. */
    MPI_Type_vector (DIMY*DIMZ, CPT_F_X[last][0], DIMX,
                     cpt_state_mpidt[stateid], &recv_strip_vec_large);
    MPI_Type_commit (&recv_strip_vec_large);
    if (NFOLDS != first_small_strip_ind[0]) {
      MPI_Type_vector (DIMY*DIMZ, CPT_F_X[0][first_small_strip_ind[0]],
                       DIMX, cpt_state_mpidt[stateid],
                       &recv_strip_vec_small);
      MPI_Type_commit (&recv_strip_vec_small);
    }               /* End if (first_small_strip_ind[0]) */

/* Receive data */
    if (0 == cmtWorldRank) {
/* First run strip then run processor, so as to traverse tmp_data2
   linearly. tmp_data2 contains the data in physical order. Going down
   the x-axis one meets first strip 0 of process 1 and then strip 1 of
   process 0 */

        i = 0;              /* Index to position in tmp_data2[] */
        tag = 0;            /* Tag for comm and request[] index */

        for (strip = 0; strip < NFOLDS; strip++) {
          int advance;    /* Bytes to advance arrays (calc taken out) */

            for (proc = 0; proc < CPT_NPROCS; proc++) {
              if (strip < first_small_strip_ind[proc]) {
                MPI_Irecv (&tmp_data2[I], 1, recv_strip_vec_large, proc,
                           tag, cmtCommCommand, &request[tag]);
              } else {
                MPI_Irecv (&tmp_data2[I], 1, recv_strip_vec_small, proc,
                           tag, cmtCommCommand, &request[tag]);
              }                   /* End if strip */
              i += CPT_F_X[proc][strip]*cpt_state_size[stateid];
              tag++;

          }                             /* End for (proc) */
        }                               /* End for (strip) */
    }                                   /* End if (0 == cmtWorldRank) */

/* Send data */
```

```
     i = 0;          /* Index to position in tmp_data[] */
     for (strip = 0; strip < NFOLDS; strip++) {
       tag = strip*CPT_NPROCS + cmtWorldRank;
       if (strip < first_small_strip_ind[cmtWorldRank]) {
         MPI_Send (&tmp_data[i], 1, send_strip_vec_large, 0,
                   tag, cmtCommCommand);
       } else {
         MPI_Send (&tmp_data[i], 1, send_strip_vec_small, 0,
                   tag, cmtCommCommand);
       }                         /* End if strip */
       i += CPT_F_X[cmtWorldRank][strip]*cpt_state_size[stateid];

     }                          /* End for (strip) */

     if (0 == cmtWorldRank) {
       if (MPI_SUCCESS != MPI_Waitall (CPT_NPROCS*NFOLDS, request,
status)) {
         fprintf (stderr, "cmt_write: MPI_Waitall failed\n");
         MPI_Abort (cmtCommCommand, -1);
       }                                /* End  (MPI_Waitall) */
     }                                  /* End if (0 == cmtWorldRank) */
```

Table 12: The code for the derived datatype and data gathering in the case of uneven decomposition. Adapted from function `cmt_write()`.

Similarly to the even decomposition case, the functions that may need to handle a subset of the substate data, `tx_vis_pack()` and `serv_view_state()` require complicated implementation. This time we discern between two cases. If the functions handle all of the data, then we do exactly what we described earlier, as shown in Table 12. Otherwise, we limit the point-to-point communication between the root process and the process holding the data; see section 7.5.3.4 for the *working process* definition. Table 13 summarises the datatype derivation.

```
/* Send types. */
    MPI_Type_vector (my_y*my_z, 1 , my_x,
                     cpt_state_mpidt[substate], &send_strip_vec_large);
    MPI_Type_commit (&send_strip_vec_large);

/* Receive types. */
    MPI_Type_vector (my_y*my_z, 1, tmp_data2_size,
                     cpt_state_mpidt[substate], &recv_strip_vec_large);
    MPI_Type_commit (&recv_strip_vec_large);

/* Receive data */
    if (0 == cmtWorldRank) {
      MPI_Irecv (tmp_data2, 1, recv_strip_vec_large, proc,
                0, cmtCommCommand, &request[0]);
    }                    /* End if (0 == cmtWorldRank) */

/* Send data */
    if (work) {
      MPI_Send (tmp_data, 1, send_strip_vec_large, 0,
```

```
                0, cmtCommCommand);
    }                        /* End if (work) */



    if (0 == cmtWorldRank) {
      MPI_Waitall (1, request, status);
    }                        /* End if (0 == cmtWorldRank) */
```

Table 13: The code for the derived datatype and gathering of a substate of the data in the case of uneven decomposition. Adapted from function `serv_view_state()`.

7.5.3.3 Reading Data

Similar operations as for writing are used when reading data. The same arrays for data storage are created per process and on process 0, although what used to serve as a receiver data store now serves as a sender and vice versa. Function in the even decomposition case is an exception to the allocation rule, as it only allocates enough space to store data the size of the *x* dimension. We will describe the even and uneven decomposition cases separately.

7.5.3.3.1 Even Decomposition Data Distribution

This time the sender (process 0) creates a fixed-extent fold type `send_fold_UB_type` deriving it from a previously derived strip vector. The receivers need only a fixed extent strip type, called `recv_strip_UB_type`, yet they receive `NFOLDS` of them and in the right order. The code is shown in Table 14 below.

```
/* Type strip */
    MPI_Type_vector (DIMY*DIMZ, CPT_F_X, DIMX, cpt_state_mpidt[stateid],
                        &send_strip_vec);
    MPI_Type_commit (&send_strip_vec);

/* Type fold */
    for (i = 0; i < NFOLDS; i++) {
      types[i] = send_strip_vec;
    }

    displacements[0] = 0;
    MPI_Address (&(tmp_data2[0]), &start_address);

    for (i = 1; i < NFOLDS; i++) {

      MPI_Address(&(tmp_data2[i*CPT_NPROCS*CPT_F_X*cpt_state_size[stateid
]]),
                  &address);
      displacements[i] = address-start_address;
    }

    for (i = 0; i < NFOLDS; i++) {
      block_lengths[i] = 1;
```

```
    }

    MPI_Type_struct (NFOLDS, block_lengths, displacements, types,
                     &send_fold_type);
    MPI_Type_commit (&send_fold_type);

/* Type fold with fixed extent for scatter */
    types[0] = send_fold_type;
    types[1] = MPI_UB;

    displacements[0] = 0;
    MPI_Address (&(tmp_data2[0]), &start_address);
    MPI_Address (&(tmp_data2[CPT_F_X*cpt_state_size[stateid]]),
                 &address);
    displacements[1] = address-start_address;

    block_lengths[0] = 1;
    block_lengths[1] = 1;

    MPI_Type_struct (2, block_lengths, displacements, types,
                     &send_fold_UB_type);
    MPI_Type_commit (&send_fold_UB_type);

/* Type strip. Different than send_ in stride since buffer is smaller */
    MPI_Type_vector (DIMY*DIMZ, CPT_F_X, CPT_DIMX,
                     cpt_state_mpidt[stateid], &recv_strip_vec);
    MPI_Type_commit (&recv_strip_vec);


/* Type strip with fixed extent for scatter */
    types[0] = recv_strip_vec;
    types[1] = MPI_UB;

    displacements[0] = 0;
    MPI_Address (&(tmp_data[0]), &start_address);
    MPI_Address (&(tmp_data[CPT_F_X*cpt_state_size[stateid]]), &address);
    displacements[1] = address-start_address;
/* The rest are the same as above */

    MPI_Type_struct (2, block_lengths, displacements, types,
                     &recv_strip_UB_type);
    MPI_Type_commit (&recv_strip_UB_type);

    MPI_Scatter (tmp_data2, 1, send_fold_UB_type,
                 tmp_data, NFOLDS, recv_strip_UB_type,
                 0, cmtCommCommand);
```

Table 14: The code for the derived datatype used for scattering data. Taken from function `cmt_read()`.

Function `serv_set_state()` does not use derived datatypes for data scattering. This function is only used when the user changes the value of one substate on one cell, despite having been implemented to handle any number of elements. In this case, the size of the allocated buffers is smaller by a factor of `yextent*zextent` because the function does not involve one-off reads from the root-process, but rather loops over the `zextent` and `yextent` to get all the data. As a result the data need to be rearranged on the receivers'

side after reception, using function `set_x_line()`. The code for scattering the data is shown in Table 15.

```
for (z = z_start; z < z_end; z++) {

      for (y = y_start; y < y_end; y++) {

        cp1 = CA_REF (ca1, z_disp+z, y_disp+y, x_disp);
        cp2 = CA_REF (ca2, z_disp+z, y_disp+y, x_disp);

        if (0 == cmtWorldRank) {

           if (size != readn (prot_sockfd, (char *) tmp_data2, size)) {
             fprintf (stderr, "serv_set_state: readn error!\n");
             MPI_Abort (cmtCommCommand, -1);
           }                        /* End if readn */

           get_scatter_ptr (&scatter_ptr, tmp_data2, tmp_data3,
                            tmp_data2_size, cpt_state_size[stateid]);

        }                           /* End if (0 == cmtWorldRank) */

        MPI_Scatter (scatter_ptr, my_x, cpt_state_mpidt[stateid],
                     tmp_data, my_x, cpt_state_mpidt[stateid],
                     0, cmtCommCommand);

        if (work) {
          set_x_line (cp1, stateid, tmp_data, my_x);
          set_x_line (cp2, stateid, tmp_data, my_x);
        }                           /* End if (work) */

     }                              /* End for(y) */

   }                                /* End for(z) */
```

Table 15: The code for scattering data without derived datatypes in the case of even decomposition. Taken from function `serv_set_state()`. Note the need to rearrange the data from normal to fold representation before reading them in the CA copies (call to `get_scatter_ptr()`).

7.5.3.3.2 Uneven Decomposition Data Distribution

Similarly to the discussion in section 7.5.3.2.2, two sizes of vectors must be defined for the sender and receivers. This operation is symmetric to the gathering, and the code reflects this too. The sender's sizes are calculated exactly like the receiver's sizes in the case of the collection and vice versa. The same symmetry appears for the strides of the datatypes. The point-to-point data communication is effected with `CPT_NPROCS*NFOLDS` immediate sends from the root process followed by `NFOLDS` standard receives from each process.

The summary of the code appears in Table 16.

```
/* Send types */
    MPI_Type_vector (DIMY*DIMZ, CPT_F_X[last][0], DIMX,
                    cpt_state_mpidt[stateid], &send_strip_vec_large);
    MPI_Type_commit (&send_strip_vec_large);
    if (NFOLDS != first_small_strip_ind[0]) {
      MPI_Type_vector (DIMY*DIMZ, CPT_F_X[0][first_small_strip_ind[0]],
                      DIMX, cpt_state_mpidt[stateid],
                      &send_strip_vec_small);
    MPI_Type_commit (&send_strip_vec_small);
    }                      /* End if (first_small_strip_ind[0]) */

/* Receive types. These are only important on the receiver side */
    MPI_Type_vector(DIMY*DIMZ, CPT_F_X[cmtWorldRank][0],
                    CPT_DIMX[cmtWorldRank],
                    cpt_state_mpidt[stateid], &recv_strip_vec_large);
    MPI_Type_commit (&recv_strip_vec_large);
    if (NFOLDS != first_small_strip_ind[cmtWorldRank]) {
      MPI_Type_vector (DIMY*DIMZ,

CPT_F_X[cmtWorldRank][first_small_strip_ind[cmtWorldRank]],
                      CPT_DIMX[cmtWorldRank], cpt_state_mpidt[stateid],
                      &recv_strip_vec_small);
    MPI_Type_commit (&recv_strip_vec_small);
    }              /* End if (first_small_strip_ind[cmtWorldRank]) */

/* Send data */
    if (0 == cmtWorldRank) {
/* First run strip then run processor, so as to traverse tmp_data2
   linearly. tmp_data2 contains the data in physical order. Going down
   the x-axis one meets first strip 0 of process 1 and then strip 1 of
   process 0 */

      i = 0;              /* Index to position in tmp_data2[] */
      tag = 0;            /* Tag for comm and request[] index */

      for (strip = 0; strip < NFOLDS; strip++) {
        int advance;    /* Bytes to advance arrays (calc taken out) */

        for (proc = 0; proc < CPT_NPROCS; proc++) {
          if (strip < first_small_strip_ind[proc]) {
            MPI_Isend (&tmp_data2[i], 1, send_strip_vec_large, proc,
                       tag, cmtCommCommand, &request[tag]);
          } else {
            MPI_Isend (&tmp_data2[i], 1, send_strip_vec_small, proc,
                       tag, cmtCommCommand, &request[tag]);
          }                    /* End if strip */
          i += CPT_F_X[proc][strip]*cpt_state_size[stateid];
          tag++;

        }                      /* End for (proc) */
      }                        /* End for (strip) */
    }                    /* End if (0 == cmtWorldRank) */

/* Receive data */
    i = 0;          /* Index to position in tmp_data[] */
    for (strip = 0; strip < NFOLDS; strip++) {
      tag = strip*CPT_NPROCS + cmtWorldRank;
      if (strip < first_small_strip_ind[cmtWorldRank]) {
```

```
        MPI_Recv (&tmp_data[i], 1, recv_strip_vec_large, 0,
                    tag, cmtCommCommand, &status[cmtWorldRank]);
      } else {
        MPI_Recv (&tmp_data[i], 1, recv_strip_vec_small, 0,
                    tag, cmtCommCommand, &status[cmtWorldRank]);
      }                           /* End if strip */
      i += CPT_F_X[cmtWorldRank][strip]*cpt_state_size[stateid];

    }                             /* End for (strip) */

    if (0 == cmtWorldRank) {
      if (MPI_SUCCESS != MPI_Waitall (CPT_NPROCS*NFOLDS, request,
                                        status)) {
        fprintf (stderr, "cmt_read: MPI_Waitall failed\n");
        MPI_Abort (cmtCommCommand, -1);
      }                           /* End if (MPI_SUCCESS != MPI_Waitall) */
    }                             /* End if (0 == cmtWorldRank) */
```

Table 16: The code for the derived datatype used for scattering data in the case of uneven decomposition. Taken from function `cmt_read()`.

The function `serv_set_state()` may handle a subset of the data. In this case it is assumed that only an *x*-plane will be distributed, and therefore one process will be reached, so only one vector datatype is constructed for the sender and one for the receiver. Data is communicated using and immediate send and a standard receive. The immediate send is obligatory to avoid a deadlock in the case that the receiver is the root process (which is also the sender). The sum of the corresponding code is shown in Table 17. In the case that the whole of the model is distributed to the processes the same code as in Table 16 is used. Note that up to release 1.3 of CAMELot this function is only used for a single cell.

```
/* Send types. */
    MPI_Type_vector (my_y*my_z, 1 , tmp_data2_size,
                        cpt_state_mpidt[stateid], &send_strip_vec_large);
    MPI_Type_commit (&send_strip_vec_large);

/* Receive types. */
    MPI_Type_vector (my_y*my_z, 1, my_x,
                        cpt_state_mpidt[stateid], &recv_strip_vec_large);
    MPI_Type_commit (&recv_strip_vec_large);

/* Send data */
    if (0 == cmtWorldRank) {
      MPI_Isend (tmp_data2, 1, send_strip_vec_large, proc,
                    0, cmtCommCommand, &request[0]);
    }                   /* End if (0 == cmtWorldRank) */

/* Recv data */
    if (proc == cmtWorldRank) {
      MPI_Recv (tmp_data, 1, recv_strip_vec_large, 0,
                    0, cmtCommCommand, &status[cmtWorldRank]);
    }                   /* End if (proc == cmtWorldRank) */
```

```
    if (0 == cmtWorldRank) {
      if (MPI_SUCCESS != MPI_Waitall (1, request, status)) {
        fprintf (stderr, "serv_set_state: MPI_Waitall failed\n");
        MPI_Abort (cmtCommCommand, -1);
      }                          /* End if (MPI_SUCCESS != MPI_Waitall) */
    }                       /* End if (0 == cmtWorldRank) */
```

Table 17: The code for the derived datatype used for scattering a substate of the data in the case of uneven decomposition. Taken from function `serv_set_state()`.

7.5.3.4 Working Macrocells and Buffer Sizes

In the case where the spatial entity concerned does not cover the full length of the *x*-axis, only some of the processes need to work in order to collect all the necessary data. Given that the CA Engine only deals with full extent entities, it is understood that these cases concern set-*x* entities (e.g. the plane *x*=1). Therefore, the data belong only to one process.

The identification of the working process is different, depending on whether the even data distribution code is enabled or not. We discuss the two cases separately.

7.5.3.4.1 Even Decomposition Working Process Identification and Buffer Allocation

This process is identified by the fact that its rank equals

$$((pos[0]-1)/CPT\_F\_X)\%CPT\_NPROCS),$$

`pos[3]` being the array denoting the position of the entity. The reason for subtracting 1 from the *x* co-ordinate is that the CA Engine enumerates the axes starting with 0, whereas the GUI and thus the user perceive the axes to start with one. Dividing by `CPT_F_X` we get the absolute number of the strip, as if the data were not spread among the processes; the modulo operation maps this to the process where it is assigned.

We will first discuss the case when a subset of data are gathered to the root process. In this case we introduce a new MPI Communicator, `cmtCommWork`, local to the function corresponding to the request in question. This communicator consists of the root process so as to do the I/O and, if the root process is not the working one, another process. If the root process is the one, `tmp_data`, `tmp_data2` and `tmp_data3` all have size

$$yextent*zextend*el\_size,$$

`el_size` being the natural size of the element; otherwise `tmp_data2` has size 2 times as much as the above. This is because `tmp_data2` must hold the data in the `MPI_Gather` call and since the root process participates in the communicator as a receiver, it also participates as a sender. It is noted that in this case the data in the first half of `tmp_data2` must be discarded. This is executed in function `get_write_ptr()`, discussed in section 7.5.3.6. This communicator is used to gather data to the root process; the size of data collected from each participating process equals the size of `tmp_data`.

In the case that data are scattered to the processes we avoid the overhead of creating and deleting the new communicator. All the processes receive the data, but each process has already determined whether the change affects its data set, using the same rule as above, and only they make the necessary changes to their CA copies.

7.5.3.4.2 Uneven Decomposition Working Process Identification and Buffer Allocation

Given that the strips have various *x*-lengths, the calculation of the working process is not straightforward in this case. The processes traverse the strips in the model comparing the *x* top end of each strip to the value of `pos[0]`. This calculation is inefficient, but it is combined with the calculation of the *x* displacement, discussed in section 7.5.3.5.2. The code is shown in Table 18.

```
  if (1 == my_x) {
    int top = 1;
    int found = 0;
    strip = 0;
    while (strip < NFOLDS) {

      proc = 0;
      while (proc < CPT_NPROCS) {
        top += CPT_F_X[proc][strip];
        if (pos[0] < top) {
          found = 1;
        }                    /* End if (pos[0] < top) */
        if (found) break;
        proc++;

      }                        /* End while (proc) */
      if (found) {
        x_disp += pos[0]-(top-CPT_F_X[proc][strip]);
        /* The distance from the current strip start */
        break;
      }                          /* End if (found) */
      x_disp += CPT_S_X[cmtWorldRank][strip];
      strip++;    /* This *must* be the last command of the loop! */
```

```
}                          /* End while (strip) */

if (proc != cmtWorldRank) {
  work = 0;
}                          /* End if (proc != cmtWorldRank) */
```

Table 18: The code for the identification of the working process in the case of uneven decomposition.

Because in the case of uneven decomposition no collective communications are used there is no longer a need for the `cmtCommWork` communicator setup; `tx_vis_pack()` is an exception, because it uses the communicator so as to calculate the minimum and maximum of the substate to be visualised (see section 8.6.2.1 for more).

Unlike the even decomposition case, `tmp_data` and `tmp_data2` have size `yextent*zextend*el_size`, `el_size` being the natural size of the element and the size of `tmp_data2` need not vary according to whether process 0 is a working process or not, because the data are communicated point-to-point. Calling the functions `get_write_ptr()`, and `get_scatter_ptr()` is not necessary either, as discussed in section 7.5.3.2.2, so the pointer `tmp_data3` is obsolete.

7.5.3.5 Data Access

Accessing the data in each of the processing elements requires knowledge of how they are stored. In the current implementation data are stored with *x* fastest as mentioned in section 5.2. As discussed there, halo data are inserted in the following places in the dataset:

- before the first and after the last real element (*z*-axis);
- between planes (*y*-axis);
- between lines (*x*-axis);
- between strips (folded data).

In the general case, the displacement in the *z*-axis equals `Radius`. This means that in order to access the first real piece of element we must skip `Radius` planes of size `CPT_Y*CPT_X` each (i.e. planes including the per-line and per-plane haloes). If the entity we want to access is not the whole model, then we must skip an extra `pos[2]-1` planes; thus, the displacement equals `Radius+pos[2]-1`. The displacement in the *y*-axis is calculated similarly.

Calculation of the *x*-axis displacement if we do not want to access the whole *x*-line is less easy. The way to do that depends on whether even decomposition is assumed or not.

7.5.3.5.1 Even Decomposition *x*-Axis Displacement Calculation

Because of the per-strip haloes, the data in the process are not contiguous; and because of the block-cyclic decomposition, they do not represent contiguous lines in the original model. In the normal case where the whole of the model is assumed, the displacement equals `Radius`, since only the initial halo in each strip must be skipped.

We will now consider the case where a subset of the data on the *x*-axis are concerned. Supposing that the right process is already located from the `cmtCommWork` communicator definition, and that the right plane and line are also located using the above rules, we have to find the correct strip in the process and the correct column in the strip and access them using a serial pointer. A `Radius` displacement will skip the line halo. In order to find the right strip we add

$$(pos[0]-1)/(DIMX/NFOLDS)*CPT\_S\_X,$$

since `DIMX/NFOLDS`[13] gives us the rank of the strip and multiplication by `CPT_S_X` takes us there. In order to find the correct column we add `(pos[0]-1)%CPT_F_X`, which gives the displacement from the beginning of the strip. In summary, if `pos[0]` is *not* equal to 0, the *x*-axis displacement equals

```
Radius + (pos[0]-1)%CPT_F_X + (pos[0]-1)/(DIMX/NFOLDS)*CPT_S_X
```

7.5.3.5.2 Uneven Decomposition *x*-Axis Displacement Calculation

As mentioned in section 7.5.3.4.2, this displacement is calculated at the same time as the working processes are identified. As shown in Table 18, the corresponding variable `x_disp` is initialised to `Radius` and then for each strip *of the process* traversed, it is incremented by `CPT_S_X[cmtWorldRank][strip]`, the total size of the strip (including the halos). However, if the working cell is found, `x_disp` is instead incremented in that process by the distance from the currently examined strip start `(pos[0]-(top-CPT_F_X[proc][strip]))`.

---

[13] We remind the reader that `DIMX` is the *x* size of the model before the decomposition, `CPT_S_X` is the total strip *x*-size including the two per-strip haloes and `CPT_F_X` is the *x*-size of the strip's real data.

7.5.3.6 Data Mapping Functions

- `void get_x_line (CptCell *cp, int substate,`
  `u_char *tmp_data, int my_x)`

  This function gets the data from the CA Engine, where they are fragmented because of the folded representation, and coalesces them into the pointer `tmp_data`. This function assumes that the cell pointer has been initialised to the first element of interest. It loops over the strips and places the data contiguously in the appropriately initialised `tmp_data` pointer passed to the function as an argument. It is executed by all the processes in the communicator when the objective is to gather substate data to the root process. The argument `substate` is used to identify the size of the elements and is also passed as an argument to `cpt_get_state()` so as to return the corresponding values. The argument `my_x` is the number of substate elements and it is used to identify whether the loop over folds should occur or there is only one element to be returned.

  _N.B.:_ This function only removes haloes, it does *not* re-organise the data so as to be contiguous for the external, natural representation of the model (see function `fold2line()` for more).

- `void set_x_line (CptCell *cp, int substate,`
  `u_char *tmp_data, int my_x)`

  This function moves the data for substate `substate` from the pointer `tmp_data` to the CA Engine copy `cp`, and at the same time it inserts haloes to the folded, yet without haloes data of the `tmp_data` pointer. This function assumes that the cell pointer has been initialised to the first byte to be written. It loops over the strips and places the data from the `tmp_data` pointer to the appropriate position in `cp` taking into account the fold-derived haloes in the latter. It is executed when substate data have been scattered from the root process to all the processes in the communicator. The argument `substate`, used to identify the size of the elements, is also passed as an argument to `cpt_set_state()`. The argument `my_x` is the number of substate elements and it is used to identify whether the loop over folds should occur or there is only one element to be set in the CA Engine.

  _N.B.:_ This function assumes that the data in `tmp_data` have been appropriately organised in folds (see function `line2fold()`).

The following functions are only used when handling parts of the model in the case of even decomposition code. Functions concerned with the whole model do not need these, as the

translation of data from normal lines to folded data and vice versa is incorporated to the corresponding scatter and gather operations. See also sections 7.5.3.2 and 7.5.3.3.

- ```
  void fold2line (const u_char *source, u_char *target,
                          size_t e_size)
  ```
  This function turns the contiguous, yet folded *x*-line data into a representation suitable for external presentation. The data are originally stored in the `source` unsigned character pointer and the resulting data are made available through the `target` pointer. `e_size` is the size of each of the elements represented as characters. The function traverses the `source` array in strides of length `NFOLDS*strip_size` and writes `strip_size` chunks of data contiguously to the `target` array. This function is only called by the root process.

- ```
  void line2fold (const u_char *source, u_char *target,
                          size_t e_size)
  ```
  This function turns the contiguous, *x*-line data into folded, internal representation data. The data are originally stored in the `source` unsigned character pointer and the resulting data are made available through the `target` pointer. `e_size` is the size of each of the elements represented as characters. The function traverses the `source` array in strides of length `CPT_NPROCS*strip_size` and writes `strip_size` (i.e. `CPT_F_X*e_size`) chunks of data contiguously to the `target` array, *without* leaving gaps for the halo. This function is only called by the root process.

- ```
  void get_write_ptr (u_char *tmp_data2, u_char *tmp_data3,
                          int tmp_data3_size, int x, int y, int z,
                          int el_size, int work, int work_size)
  ```
  This function is a wrapper[14] for `fold2line()`. The argument `tmp_data2`[15] contains the original data in internal CA Engine format, and `tmp_data3` is the target buffer for `fold2line()`. It is assumed that the former has been appropriately initialised to contain folded data, whereas the latter points to an appropriately allocated memory block of sufficient size to hold data for the whole of the automaton. `el_size` is the natural size of the elements stored as unsigned characters in `tmp_data2`. The variable `tmp_data3_size` is the size of the array `tmp_data3` in the *x* dimension. The function loops over `z` and `y` in that order and sets `x` elements of `tmp_data3` each time. There are two cases for `tmp_data3_size`:

---

[14] The implementation of this function has changed radically since release 1.0 of the software.
[15] It may help the reader to note that we maintained the naming of the variables of the calling function (see section 7.5.3.1).

- If `tmp_data3_size` equals `DIMX` then all the processes are working, therefore
  `fold2line(tmp_data2, tmp_data3, el_size)` is called;
- If it equals 1, then there is no need for data rearrangement (they are just an element)
  `el_size` bytes are copied from `tmp_data2` to `tmp_data3`.

There is a slight complication though, which justifies the existence of the `work` and `worksize` arguments. The former is a flag denoting whether the root process was the only member or if there was another process in the communicator. In the latter case, the size of `tmp_data2` is 2 and the data in the first half of `tmp_data2`, originating from the root process, must be discarded since the second process contributed the correct data. This is achieved by advancing the `tmp_data2` pointer by `el_size` before entering the loop.

After each iteration, the source and target pointers must be advanced. The argument `work_size` contains the size (number of processes) of the communicator, and the argument `x` is the size of each strip. Therefore, after each iteration `tmp_data2` is advanced by `x*work_size*el_size` bytes and `tmp_data3` is advanced by `tmp_data3_size*el_size` bytes.

When the function exits, the argument `tmp_data3` points to the rearranged data, suitable for the external representation of the system.

- `void get_scatter_ptr (u_char **scatter_ptr, u_char *tmp_data2,`
  `u_char *tmp_data3, int tmp_data2_size,`
  `int el_size)`

  Similarly to `get_write_ptr()`, this function is a wrapper for `line2fold()`. It returns the pointer `scatter_ptr` (passed *by reference*) containing data in internal, folded representation. The argument `tmp_data2` contains the original data in internal CA Engine format, and `tmp_data3` is the target buffer for `line2fold()`. `el_size` is the natural size of the elements stored as unsigned characters in `tmp_data2`. Note that we now use the variable `tmp_data2_size` which is the size of the array `tmp_data2`. There are two cases for it, *not* three (`DIMX` or 2 or 1) as is the case when writing data, since when reading data there is no reason to allocate extra space for the root process, `tmp_data2size` is the number of data to be scattered to *each* process.

- If `tmp_data2_size` equals `DIMX` then all the processes are working and the function calls `line2fold(tmp_data2, tmp_data3, el_size)` and assigns `*scatter_ptr` to point to `tmp_data3`.
- If it equals 1, then there is no reason for data rearrangement (they are just an element); `*scatter_ptr` is set to point to `tmp_data2`.

N.B.: `get_scatter_ptr()` differs from `get_write_ptr()` in that the former concerns the whole of the model, whereas the latter is only applied to one *x*-line only.

7.5.4 Substate Related Functions

- `int serv_save_request (void)`
  This corresponds to the `req_save_request()` GUI function which requests that the `substate` values for all cells in the CA Engine be written to file `filename`. After reading `substate` and `filename` from the socket, the root process broadcasts the state id to all the cells. The function `cmt_write` (section 7.5.8.1) is called then, to perform the write to file.

- `int serv_set_load (void)`
  Set the `substate` values for all cells in the Engine to those listed in file `filename`. This is the inverse function of `serv_save_request`. The same procedures as above are followed and then the function `cmt_read` (section 7.5.8.1) is called to read the data from the file and update the CA copies.

- `int serv_view_state (void)`
  This function writes the data of a subset of the model to the socket. The root process on the CA side reads the `pos[]` array containing the co-ordinates of the entity to be retrieved as well as the `substate` id through the socket. The array and substate are broadcast to all the processes which calculate the loop extends as well as the temporary data storage size as described in section 7.5.3.2. Then the working cells are identified as described in section 7.5.3.4 and the participating processes allocate the temporary memory buffers. After the displacement has been calculated the working processes loop over *z* and *y* and the root process gathers and rearranges the data as described in section 7.5.3.2 using derived datatypes as shown in Table 11, Table 12 and Table 13. The pointer is advanced by calling `CA_REF` after every *y* iteration, rather than by advancing the pointer using pointer arithmetic, as is the case when the whole model is being handled (e.g. in function `cmt_write()`). Process 0 then writes back to the GUI the current generation and executes one `writen` call to write the data to the

socket. After completion of the task, the communicator, the derived datatypes and the temporary memory buffers are freed.

- `int serv_set_state (void)`
  This handles the request that a substate of the entity in a plane array `pos[]` be set to the value transmitted through the socket. The root process reads and disseminates the details of the entity in question from the socket. The flow of the program differentiates with respect to whether even decomposition is chosen.

  In the case of even decomposition the processes allocate temporary buffers on a per-line basis. Each process also determines whether it needs to work and defines the loop extends. In contrast with `serv_view_state()` which needs one in the case of even decomposition, no communicator is necessary. Inside the nested loop two copies are accessed. The root process reads the data in *x*-line portions from the socket and scatters them as shown in Table 15, after calling `get_scatter_ptr()` to rearrange the data on a per-line basis. All the processes that need to work call `set_x_line()` twice to update their CA array copies. Note that this function calls `get_scatter_ptr()` as many times as the loop iterations and `set_x_line()` twice as many times. The loops are shown in Table 15.

  In the case of uneven decomposition, memory for data storage is allocated, as described in section 7.5.3.4 for the full extent of the data. This effects only one `readn` call to read the data from the socket. The send and receive vectors are created as described in section 7.5.3.2.2 and shown in Table 16 and Table 17. As shown in Table 15, the data are written to the CA copies line by line.

  In both cases, the temporary buffers are freed in the end.

- `int serv_set_param (void)`
  The root process reads the number of parameters to be set and aborts if the number is illegal. It then loops over `no_of_params` reading the index and setting the value of the corresponding parameter. After this is done, the internal parameter array is broadcast to all the cells.

7.5.5 Program Flow Management

The PAUSE, LOOP, EVOLVE, FINISHED, EXITCODE and RESUME `req_codes` are implemented inside the `rv()` function (section 7.5.2).

- `int serv_terminate (void)`

  When TERMINATE is called the visualisation and plane lists are deleted. This is discussed in the Visualisation section.

- `int serv_set_fold (void)`

  The root process reads the starting and ending active fold index in an integer array with two elements, which it broadcasts to the other processes. If the specified folds are invalid (i.e. not in the correct order or not in the range [0, NFOLDS-1]) the function returns IGNORED. If the start and end folds are 0 and NFOLDS-1 respectively, then the manual folds are terminated and the automatic inactive strip detection mechanism is set. Otherwise, the mechanism is deactivated and the `active_strip[]` internal array is updated according to the active fold specification.

7.5.6 *Visualisation Functions*

- `int serv_add_plane (void)`
- `int serv_del_plane (void)`
- `int serv_set_minmax (void)`

The functions concerned with the visualisation are discussed in the Visualisation section.

7.5.7 *Configuration (Project) Related Functions*

- `int serv_proj_read (void)`

  The `filename` read through the socket is used as a root for the files to be read. The root process truncates the extension of the filename and does not broadcast it to the other processes as they do not need it. They all call `cmt_read_all()` discussed in section 7.5.8.1.

- `int serv_proj_save (void)`

  The root process reads the pathname of the files to be written. All processes call `cmt_write_global()` and `cmt_write()`, emulating `cmt_write_all()` behaviour excluding the AVS field file functionality and index keeping. See section 7.5.8 for more details on the functions previously mentioned.

- `int serv_periodic_save (void)`

The root process reads the pathname of the files to be saved periodically, as well as the period, `save_step`. The former is turned into a pathname and a filename stored respectively in the global variables `out_dirname` and `out_basename`. The filename is checked against the filesystem, by means of the function `check_fs()`, looking for already existing files which could be overwritten. The result of this search is written back to the GUI, and can be either `OVER_W` if there are such files, or 0 (zero) if there are not. The `save_step` is broadcast to all the processes and used by the `run` function, unlike `out_dirname` and `out_basename` which are not needed in the other processes. Periodic saving is handled by function `run()`, as mentioned in section 5.1.2.7.

7.5.8 Auxiliary Functions

7.5.8.1 File I/O related

The following functions return 0 if execution is correct; otherwise, a negative value is returned.

- `int cmt_read_global (char *filename)`
  The root process reads a binary file containing all global CA information for the current generation. The binary file `filename.cpj` is needed for the function to work. It contains data concerning the following:

  – The dimension of the automaton;
  – The *x*, *y*, *z* dimensions of the model;
  – The current generation;
  – The number of states;
  – The number of folds;
  – The number of global parameters and their values.

  The data are collected in an eight element integer array with the exception of the parameter values which are stored in the appropriate array and are then broadcast to all the processes. The function checks the correctness of the above values (except for the generation and parameter values) against the ones already set and sets the generation (the parameter values are set during the broadcast).

- `int cmt_read_all (char *filename)`

  This function calls `cmt_read_global(filename)` in order to read the global parameters. It then calls `cmt_read` to handle the substate files. Let *n* be the number of substates. The CA Engine expects the existence of *n* binary files named `filename[TLC].cmt`, `[TLC]` being a three-digit numerical identifier for each of the substates. For example, the first substate will be associated with the file `filename000.cmt`.

- `int cmt_read (char *filename, int substate)`

  Given that all of the CA Engine data on the substate are to be read from the file, the temporary data storage structures are allocated maximum memory, as discussed in section 7.5.3.2. The root process opens the file designated by `filename` and reads the data using only one call of the appropriate function, depending on whether XDR is used or not (see section 5.2.2 for the use of XDR in CAMELot). The data are then scattered to the processes using as few MPI calls as possible, as described in section 7.5.3.3 and shown in Table 14. In order for the processes to update their local data two `CptCell` pointers are used, pointing to each of the two CA array copies because the changes should be applied to both of them. The original displacements are minimum (`Radius` on each axis). The processes loop in parallel over *z* and *y* calling `set_x_line()` for both copies. After each *y* loop the CA pointer is advanced by `CPT_X`, which is a line including fold and line haloes, and after each *z* loop it is advanced by `2*Radius*CPT_X`, which is a plane halo at the end of the current plane and a plane halo at the beginning of the next plane. The temporary buffers and derived datatypes are freed on exit from the function.

- `int cmt_write_global (char *filename)`

  The root process writes a binary file containing all global CA information for the current generation. The binary file `filename.cpj` is created. The data it contains are the same as those that `cmt_read_global()` expects to read. Because the data written to the file are global, this function performs no MPI communications.

- `int cmt_write_all (char *dname, char *bname)`

  Saves the global and substate data in files named using an index incremented every time the function is called (static variable). It also updates the related AVS/Express field file.

  The global data are stored in a file named `dname/[TLC]bname.cpj`, `[TLC]` being a three-digit numerical identifier for the index, thus allowing for 1000 consecutive saves

before overwriting the initial file. This is done without warning the user[16]. The function calls `cmt_write_global()` in order to write the global parameters.

The function then loops over the substate ids, calling `cmt_write()`, the filename following the above convention for prefixing the filename and the same convention as in `cmt_read()` to handle substate files. For example, the first save of the first substate will be associated with the file `dirname/000filename000.cmt`.

If the function is called for the first time, `cmt_create_fld()` is called to create the necessary AVS/Express field files. The `cmt_write_fld()` function is then called to update the contents of the field file. Both these functions are explained below.

- `int cmt_write (char *filename, int substate)`
  The function writes to the file `filename` the data for state `substate`. Given that all of the CA Engine data on the substate are to be written to the file, the temporary data storage structures are allocated maximum memory. The data are accessed through a pointer to the CA Engine. The original displacements are minimum (`Radius` on each axis). All the processes then loop over *z* and *y* executing `get_x_line()`, collecting the data in `tmp_data`. After each *y* loop the CA pointer is advanced by `CPT_X`, which is a line including fold and line haloes, and after each *z* loop it is advanced by `2*Radius*CPT_X`, which is a plane halo at the end of the current plane and a plane halo at the beginning of the next plane. After the loop is finished the data are gathered in the `tmp_data2` pointer of process 0 (see section 7.5.3.2 for more details on the gather strategy). The root process writes the data to the file with one call, using XDR primitives if so selected by the user. The temporary buffers are freed on exit from the function.

- `int cmt_create_fld (char *dname, char *bname)`
  This function creates an AVS/Express field file for each datatype of the substates. It also writes the initial data containing the specification of the simulation (i.e. all the data appearing before the first line tagged `time`). The format of these files appears in Table 19. It uses the variable `dt_list`, of type `state_dt_list` (see section 5.1.1.4) to loop over the various datatypes of the substates and create one field file for each of them. The arguments of this function are used as described in the discussion of `cmt_write_all()`.

---

[16] Warning against overwriting existing files is generated when assigning the filename for a periodic operation. See section 7.5.7 for further details.

The `label` field lists the names of all the states of a given datatype, expanding array states (so `myarr[`*n*`]` is expanded to `myarr[0] myarr[1] ... myarr[`*n*`]`). Because AVS does not accept the use of brackets (`[` and `]`), these are replaced bu underscores (the character '`_`'). Therefore for the example above the expanded list is `myarr_0_ myarr_1_ ... myarr_`*n*`_`. In order to avoid matching the modified names with those of scalar variables, scalar variable names are postfixed with the underscore character.

- `int cmt_write_fld (char *dname, char *bname, int time)`
  This function loops over the `statetypes` members of `dt_list` and for each of them it loops over their states. It thus accesses the state type and index for each of the substates and adds the `variable` entries to the appropriate field files. Using the above strategy, each field file is opened only once during a call to the function. The arguments of this function are used as described in the discussion of `cmt_write_all()`.

```
# AVS field file
# CAMELot generated
nstep = <number of expected¹⁷ saves>
ndim = <model dimension>
dim1 = <x-dimension>
dim2 = <y-dimension>
dim3 = <z-dimension>
nspace = 3
veclen = <number of associated substates>
data = <datatype of associated substates>
field = uniform
label = <names of associated substates>

time value = 1
variable 1 file = <filename> filetype = binary
variable 2 file = <filename> filetype = binary

...
EOT¹⁸

time value = 2
...
```

Table 19: Format of CAMELot Generated AVS Field Files

---

[17] This could differ from the number of actual saves if the user ends the run prematurely

[18] Starting with release 1.3 of the software, the `EOT` separator appears between blocks of data refering to consecutive time steps

7.5.8.2 `state_dt` and `state_dt_list` Related

- `void init_state_dt (state_dt *st_dt_ptr, MPI_Datatype data)`
  This initialises the `states` member of `st_dt_ptr` to 0 and the `data` member to `data`.

- `int add_state (state_dt *st_dt_ptr, int stateid)`
  This adds the substate `stateid` to `st_dt_ptr` and increments its `states` member. It also performs checks to `stateid` and its datatype as well as to `states`. If the checks fail it returns -1, else it returns 1.

- `void init_state_dt_list (state_dt_list *st_dt_l_ptr)`
  This initialises the `many` member of `st_dt_l_ptr` to 0 and loops over the states of the system calling `add_state_dt()`.

- `int add_state_dt (state_dt_list *st_dt_l_ptr, int stateid)`
  This first searches the `statetypes[]` member of `st_dt_l_ptr` for an element with the same `data` field as `stateid`. If it does not find one, it calls `init_state_dt()` augmenting the active range of `statetypes` and increments the `many` member. It then calls `add_state()` to add the state to the `state_dt` found. It also performs checks to `stateid` as well as to `many` and the return value of `add_state()`. If the checks fail it returns -1; otherwise, it returns 1.

8.   Visualisation

The CA Engine transmits periodically substate data to the GUI. Although the GUI defines the planes and visualisation steps, this and `GEN_NO` are the only situations in which the CA Engine initiates the transmission of data. Although the transmission follows the same procedure as any output to a file or socket, the implementation of the visualisation functionality required the introduction of various data structures on the GUI and the CA Engine. The protocol for the maintenance of the visualisation entities is slightly complicated because of the variety of possible events. Moreover, a colour mapping strategy was devised.

8.1 Data Structures

8.1.1 Plane Definition

A plane in the CA context is generally defined by:

- Two Cartesian triples defining points in the CA space;
- A substate to be visualised;
- A visualisation step;
- A plane ID, unique to the system (i.e. a plane should be referred to by the same ID in both the GUI and the CA side).

The convention for the spatial extent of the plane above can denote anything from a cube to a point in the CA space. We decided to consider 2-D planes as the finest granules of the visualisation procedure[19].

In the initial release of the engine we only implement full extent planes, i.e. 2-dimensional spaces occupying maximum area. We thus use *only one* point in space, the co-ordinates of which should be zero except for one co-ordinate which should be greater than zero and less than the maximum dimension. For example, (0,3,0) denotes a *y*-plane in position 3 if the dimension of *y* is 3 or more. On the other hand, (-1,0,0) and (1,2,1) are illegal (the latter generally denotes a point).

After the above discussion we introduce the following type definitions.

---

[19] With the exception of 1-D models.

```
typedef struct {

        int pos[3];

        } point;
```

The `pos[]` array holds the co-ordinates of the point defining the plane. It represents a 3-D triple and the first field holds the *x* co-ordinate, the second the *y* and the third the *z*.

```
typedef struct {

        point pt;

        int substate;

        int vis_step;

        int ID;

        plane_class *class_ptr;

        } plane;
```

The `pt` member holds the spatial identity of the entity; `substate` is the visualised substate id; `vis_step` is the period of visualisation for the plane; `ID` is a unique identifier for internal plane representation and handling; `class_ptr` is a pointer to the `plane_class` structure holding the class information for the plane in question. Plane classes are discussed next.

8.1.2 Plane Classes

Because of the way the plane was defined, two planes extending in the same area visualising the same substate will be considered different if they differ in the visualisation step. As a result, the data for the plane will be sent more than once to the GUI if the current CA Engine generation is divided by the visualisation steps of more than one plane of the above described kind.

We therefore introduced the idea of a plane class, linking such planes with the last visualisation performed. To implement this we introduce this type definition:

```
typedef struct {

        int no_planes;

        int last_vis;

        } plane_class;
```

The member `no_planes` denotes the number of planes in the class; `last_vis` is the latest
CA Engine iteration when there has been a visualisation of a plane in the class. It should be
noted that plane classes are *not* maintained in the GUI side planes.

8.1.3 Plane Lists

Both sides of the system maintain a list of all the planes visualised. We introduced the fol-
lowing data structure for the purpose.

```
typedef struct {
        plane **planes;
        int no_planes;
        int max_index;
        int size_of_list;
        } plane_list;
```

The `planes` member is the array of `plane` pointers we want to maintain; `no_planes` is
the number of planes currently in the list; `max_index` is the number of planes added to the
list since its initialisation; `size_of_list` is the dimension of the `planes` array.

Plane lists play a most important role in the addition and deletion of planes.

8.1.4 Visualisation List

The CA Engine maintains a sorted list of visualisation generations containing exactly one
entry for each plane. This list is used to check whether the state of a plane must be trans-
mitted and to get a handle to this plane. The cells of this list have the following form:

```
typedef struct _cell_ {

        plane *data;

        int generation;

        struct _cell_ *next;

        struct _cell_ *prev;

        } cell;
```

The `data` member is a pointer to the plane; `generation` is the next visualisation genera-
tion of the plane; `next` and `prev` are links to the next and previous members in the list.
The list is then implemented as a type:

```
typedef struct {

        cell *head;

        cell *tail;

        int size;

        } list;
```

The first two members are pointers to the ends of the list; `size` is the number of elements
in the list. A plane enters this list when introduced to the CA Engine and it is removed
from it when a `DEL_PLANE` request is issued with its ID. This data structure plays a central
role in the visualisation process.

8.2 Global Variables

8.2.1 CA Engine Global Visualisation Variables

- `list vis_list;`
  A list of all the visualisation planes, maintained in ascending order with respect to the
  CA Engine iteration when each will be visualised next.

- `plane_list all_planes;`
  A list of all the planes in the CA Engine.

- `double minmax[NumOfStates][2];`
  The minimum and maximum value for each substate (updated only if the substate is visualised and only with the union of the data subsets visualised). `minmax[][0]` holds the minima and `minmax[][1]` holds the maxima.

8.2.2 GUI Global Visualisation Variables

- `int nvizwins`
  Number of currently-open Visualisation windows.

- `VIZWIN *vizwins[]`
  A fixed-size array of pointers to all currently-open Visualisation windows' `VIZWIN` structures. When a Visualisation window is closed, the memory for the `VIZWIN` structure is released and the corresponding `vizwins[]` pointer set to `NULL`, although the array element is not reused until the Simulation window is exited.

- `VIZWINLISTNODE *plane2win[]`
  In order to map visualisation planes received from the CA Engine to Visualisation windows, a linked list of pointers to `VIZWIN` structures is maintained for every currently-visualised plane. The head node of each list is pointed to by a fixed size array of pointers (`plane2win[]`) indexed by plane ID.

- `int planerefcnt[]`
  Used to keep a reference count of windows for each plane. When the reference count for a plane reaches 0, i.e.., no window now shows this plane, a `DEL_PLANE` request is sent to the CA Engine.

- `plane_list all_planes`
  Similarly to the `all_planes` variable in the CA Engine this is a list to all the planes in the GUI.

- `buffer viz_buffer`
  The buffer for visualisation plane reception.

8.3 Relevant Files and Functions

8.3.1 File `common.h`

Contains the declarations of the following (as well as others, not related to visualisation):

* `point` type and the respective functions (`plane.c`);
* `plane` type and the corresponding functions (`plane.c`);
* `plane_list` type and the related functions (`plane.c`);
* `cell` and `list` types and their functions (`list.c`);
* `buffer` type and functions (`buffer.c`).

8.3.2 Files `guicomms.h` and `guicomms.c`

Contain the declarations and implementations of the visualisation-related functions of the protocol discussed next. Additionally, the following functions are contained in the files.

* `int consume_vis_pack (void)`

  Consumes visualisation packets from the visualisation socket. It is used to remove obsolete visualisation packets when an event which stops normal execution occurs. It is implemented by means of a loop over `select(3C)` on the visualisation socket. If there is a visualisation message, `rv_vis_pack()` writes the data to a suitably initialised `buffer` (see section 8.6.3.1 for more). This static buffer is allocated memory once throughout the program life, when `consume_vis_pack()` is first called.

* `void GUI_check_pos (int *pos)`

  Checks `pos` against `xyzdims[]` to correct unacceptable values. Correction is done by setting the coordinate to 0. When the size of a dimension of the model is 1, it sets the corresponding coefficient to 1 (rather than 0) to prevent identifying planes as cubes. For example, (0,0,0) in a 2-D model will be turned to (0,0,1) which is a plane.

* `int GUI_get_val_size (const int *pos)`

  Returns the number of elements specified by `pos[]`. This is done by multiplying the assumed size (originally 1) by the size of the model's dimension if the corresponding coefficient in `pos[]` is equal to 0.

- `int get_max_size (const unsigned int *pos)`
  Returns the maximum of the possible products of 3 choose 2 elements of the 3-element array `pos[]`. It is used to derive the maximum possible number of elements for the visualisation buffer, taking as its argument the array `xyzdims[]`. It calculates the three possible sizes and returns the maximum.

8.3.3 File `macrocell.c`

Contains the declarations and implementations of the visualisation-related functions listed in section 5.1.2.5 and further discussed in this section. It also contains the following functions:

- `static int tx_vis_pack (cell *, char)`
- `static void colour_map (const u_char *, u_char *, int, int,`
  `                        double, double)`

These are discussed later in this section.

- `static void check_pos (int *pos)`
  Same as `GUI_check_pos()`, only that it checks against `DIMX`, `DIMY`, `DIMZ`, instead of `xyzdims[]`.

- `static void check_plane (plane *pl_ptr)`
  This function checks and corrects the plane for spatial, substate and visualisation step consistency. Calls `check_pos()` for the array consistency and makes a separate check if the model is 1-D. If the plane is found illegal it sets its `ID` member to `IGNORED`, otherwise it sets it to -1.

- `static void bcast_plane (plane *pl_ptr, MPI_Comm comm)`
  This function broadcasts the details of the plane as detailed in process 0 of the CA Engine to all the processes in the communicator. It does not set up a new datatype containing the 5 integers which are broadcast (i.e., `pl_ptr->pt.pos[3]`, `pl_ptr->substate`, `pl_ptr->vis_step`).

- `static int get_val_size (const int *)`
  Similar to `GUI_get_val_size()`.

8.3.4 File `plane.c`

8.3.4.1 Related to `point`

- `void init_point (point *pt_ptr, int x, int y, int z)`
  Initialises the point passed as an argument by reference with the given coefficients.

- `int write_point (int sockfd, const point *pt_ptr)`
  Writes the point coefficients to the socket; calls `writen()` only once. Returns 0 if `writen()` succeeds, -1 otherwise.

- `int read_point (int sockfd, point *pt_ptr)`
  Similar to the above, only that it reads the point data.

- `int ptcmp (const point *cp_ptr1, const point *cp_ptr2)`
  Loops over the coordinates and compares the coefficients of the two points. Returns 0 if they are equal, 1 otherwise.

8.3.4.2 Related to `plane_class`

- `void init_pl_class (plane_class *cl_ptr)`
  Sets `no_planes` to 0, `last_vis` to -1.

- `del_pl_class (plane_class **cl_ptr_ptr)`
  This decrements the `no_planes` member of the pointer to a `plane_class` to be deleted, and if this then equals zero, the pointer is freed; thus the reason for passing it by reference.

8.3.4.3 Related to `plane`

- `void init_plane (plane *pl_ptr, const point *pt, int substate,`
  `                  int vis_step)`
  Sets the corresponding members of the plane pointed by `pl_ptr` to those passed as arguments. The `ID` is set to 0 and the `class_ptr` is set to `NULL`.

- `void disc_plane (plane **pl_ptr_ptr)`
  This function first calls `del_pl_class()` to delete the `class_ptr` member of the `plane` struct and then frees the memory for the `plane` pointer passed to the function *by reference*.

- `int write_plane (int sockfd, const plane *pl_ptr)`
  Calls `write_point()` and then writes the `substate` and `vis_step` members of the plane to the socket. It does *not* write the `ID`, or the `class_ptr` details, which are assigned separately on each side during the plane addition process.

- `int read_plane (int sockfd, plane *pl_ptr)`
  Similar to `write_plane()` in action and behaviour.

- `int plcmp (const plane *cp_ptr1, const plane *cp_ptr2,`
               `plane_class **pl_c_ptr)`
  The function checks the two planes pointed by the constant pointers for equality of the `substate`, `vis_step` and `pt`[20] members. Moreover, if the `pl_c_ptr` plane class *pointer-pointer* argument passed to the function is *not* 0, then the function performs a check to find which of the two planes already belongs to the plane list and returns a handle to its plane class through `pl_c_ptr`. This suggests that the GUI-side caller function *must* pass the argument as 0.

  The function returns:

  - 0, if the two planes are equal;
  - 1, if the two planes are in the same class;
  - -1, otherwise.

8.3.4.4 Related to `plane_list`

- `void init_plane_list (plane_list *pl_l_ptr)`
  It allocates space for the `MAXPLANES plane*` elements of the `planes` array member of the structure. Sets `size_of_list` to `MAXPLANES`, `no_planes` and `max_index` to 0 and zeroes the pointers in the `planes` member.

- `void clear_plane_list (plane_list *pl_l_ptr)`
  Removes all planes from a `plane_list`, without deleting it. It calls `disc_plane()` to discard each plane. It zeroes `planes[i]`, `max_index` and `no_planes`, thus returning the list to the state where `init_plane_list()` leaves it.

---

[20] It uses the trivially implemented `ptcmp()` function to this end.

- `int add_plane (plane_list *pl_l_ptr, plane *pl_ptr,`
  `                 int *ID_same, int ch_class)`
- `int rem_plane (plane_list *pl_l_ptr, int ID)`

These are discussed extensively in the paragraphs about plane addition (8.4.2) and deletion (8.5.2).

8.3.5 File `list.c`

8.3.5.1 Related to `cell`

- `void init_cell (cell *c_ptr, const plane *pl_ptr,`
  `                 int generation)`
  This function zeroes the forward and backward pointers (`next` and `prev`) and sets the `data` and `generation` members to those passed as its arguments.

- `static void del_cell (cell **c_ptr_ptr)`
  First calls `disc_plane()` to discard the plane in the `data` member and then frees the memory occupied by the cell. This function is not publicly available.

8.3.5.2 Related to `list`

- `void init_list (list *l_ptr)`
  Zeroes the `head`, `tail` and `l_size` members.

- `void set_gen (list *l_ptr, int gen)`
  Resets the next visualisation generation of all the cells in the list to `gen+1` and sets `last_vis` in all the plane classes of the planes in the respective `data` members to -1. These two actions cause the planes to be visualised immediately. It is used when restarting the CA Engine and it assumes that the CA Engine iteration index is set to `gen`.

- `void clear_list (list *l_ptr)`
  This function deletes the cells of the list, but assumes that the plane members have already been deleted. It does not delete the list itself.

- `cell *first (list *l)`
  This function provides a pointer to the `head` of the list, or `NULL` if the list is empty.

- `int del_ID (list *l_ptr, int ID)`

  This searches the doubly-linked list for the `cell` containing the plane with the given `ID`. It removes this from the list and then calls `del_cell()` to discard the plane and free the cell's memory. `del_ID` returns `DEL_PLANE` if the plane is found or `IGNORED` else.

- `void reorder (list *l_ptr, cell *c_ptr)`

  This function removes the `cell` pointed by its second argument and reinserts it in ascending order with respect to its `generation` member. After amending the `next` and `prev` pointers of the cell's previous and next neighbours respectively, the function calls `insert()` for the actual reinsertion.

- `void insert (list *l_ptr, cell *c_ptr)`

  The function inserts a cell in the list so as to maintain ascending order of the cells with respect to their `generation` member. It makes use of three trivial internal functions, namely `addhead()`, `addtail()` and `addmiddle()`.

8.3.6 File `buffer.c`

A datatype we have not previously discussed is the `buffer`. It is used by the visualisation functions on the GUI side so as to enable one-off memory allocation for each of the planes visualised. Its declaration is as follows:

```
typedef struct {
        u_char *data;
        int size;
        } buffer;
```

There are two functions associated with this structure:

- `int init_buffer (buffer *buf_ptr, int size)`

  This function allocates `size` bytes of memory for the member `data` and sets the `size` member. It returns -1 if `malloc` fails or `size` is less than 1; otherwise it returns 1.

- int expand_buffer (buffer *buf_ptr, int size)

  If the newly defined `size` is greater than the `size` member of the structure it uses `re-alloc` to expand the `data` member and resets `size`. It returns -1 in case of failure, 1 otherwise.

8.4 Plane Addition

8.4.1 Addition Protocol

Plane addition is initiated by the GUI. It sends the point defining the location of the plane, substate to be visualised and the visualisation step to the CA Engine (i.e. it transmits a plane without an ID and a plane class pointer, using the `write_plane()` library function) through the communication abstraction. This communication is performed through the usual `prot_sockfd` socket. The CA Engine replies with the ID of the plane and acknowledges addition. The *normal case* protocol is shown below:

Sender	Token	Type
GUI	ADD_PLANE	req_code
GUI	pos[3]	int *
GUI	substate	int
GUI	vis_step	int
CA	ID	int
CA	ADD_PLANE	req_code
GUI	VIS_PACK	req_code

As we discuss next, the protocol is more complicated in the cases of adding an already existing plane.

8.4.2 The Function add_plane() and Other Related Functions

The desired effect is to add the plane pointed by `pl_ptr` to the plane list pointed by `pl_l_ptr`. The prototype of the function is as follows:

```
int add_plane (plane_list *pl_l_ptr, plane *pl_ptr, int *ID_same,
               int ch_class)
```

The function is called from the plane addition functions of both the CA Engine and the GUI. The last two arguments differentiate between the two cases. We note that the GUI-side caller should pass zeroes (0) in the last two arguments, and defer the discussion for later in this section. The behaviour of this function describes the plane addition strategy.

The function traverses the plane list searching for a plane which is exactly the same as the one we want to add or belongs to the same class. In the case of the GUI, because the plane classes are not maintained, the plane class check is not performed. This is denoted by means of the `ch_class` flag which should be cancelled if the caller is on the GUI side.

The plane comparison is performed by the function `plcmp`. If it returns 0, then a `NULL` plane pointer is added to the list, occupying the position and index. The `ID` member of the plane pointer is updated with the *negated* value of the ID that the plane would have if it had been added. Moreover, if the `ID_same` argument is not set to zero (i.e. the caller is the CA Engine), then the ID of the plane that was found to be equal in the list is returned through the argument. In this case the function returns `IGNORED`, exiting immediately.

If `plcmp()` returns 1 or -1, then the search in the list is continued. In the former case the plane class pointer returned through the `pl_c_ptr` argument of `plcmp()` is stored. On exiting the list traversal, the function adds the plane to the list and sets its `ID` field to the value of the `max_index` member of the list. The `max_index` and `no_planes` members of the list are then incremented. If the `ch_class` flag is set and no plane in the same class has been found, a new plane class instance is created. Its `no_planes` member is set to 0, but its `last_vis` member is set to –1 by means of the `init_pl_class` function. On the other hand, if a plane class address has been stored during the traversal, the `class_ptr` member of the plane being added to the list is set to what that address points to and the corresponding `no_planes` member is incremented. The possible combinations of the re-turn value with `pl_ptr->ID` are shown in Table 20.

Case	*Return*	`ID`
Successful addition (in existing class or not)	`ADD_PLANE`	`pl_l_ptr->max_index`
`pl_ptr` already in list	`IGNORED`	`-(pl_l_ptr->max_index)`
`malloc` or other failure	`-1`	*<Undefined>*

Table 20: Combinations of the return value of `add_plane()` and the `ID` of the plane

8.4.3 GUI-Side Plane Addition

Addition on the GUI side is handled by the following function:

```
int req_add_plane (plane *pl_ptr, int *ID_same)
```

The function implements the protocol, by sending the data of the plane pointed to by `pl_ptr`. There are two possibilities for the `ID` it then reads. If it is `IGNORED`, then this means that the plane has been discarded on the CA Engine side. In this case the function immediately returns the value 0, emulating the behaviour of `get_ack()` when the latter receives `IGNORED`. If the `ID` is not `IGNORED`, it can still be negative, in the case that the plane already existed in the CA Engine. The function calls `add_plane()`, which contains all the necessary data to see if the plane already exists. The difference is that the `ID_same` and `ch_class` arguments of `add_plane()` must be passed zero, as discussed previously. The id received through the socket is checked against `pl_ptr->ID` which is set inside `add_plane()` to ensure consistency between the two sides. Finally, if `add_plane()` returns `IGNORED`, `ID_same` is read from the socket and 0 is returned; otherwise, `get_ack()` is called with (effectively) `ADD_PLANE` as an argument and its return value is returned by `req_add_plane()`.

The possible combination of the return value, the id assigned to the plane and the id of the same plane found in the CA Engine (when applicable) are given in Table 21 below.

Case	Return	ID read from socket	ID_same (socket)
Successful addition	get_ack(ADD_PLANE)	>= 0	<not read>
pl_ptr illegal	0	IGNORED	<not read>
pl_ptr already in list	0	< 0	>=0
other failure	-1	<Undefined>	<not read>

Table 21: Combinations of the return value of `req_add_plane()`, the `ID` and `ID_same` read from the socket

8.4.4 CA Engine-Side Plane Addition

This is handled by the following function:

```
int serv_add_plane (void)
```

The root process of the CA Engine reads through the socket the details of the plane to be added and creates the plane without the ID. Function `check_plane()` uses the `ID` field of the newly-defined plane to identify an illegal plane by setting it to `IGNORED`. The other processes call `bcast_plane()` to get the details of the plane. The following, with the exception of the communication with the GUI, happen to all the processes.

If the plane definition is acceptable, `add_plane()` inserts it in the list, sets its `ID` again, and also sets `ID_same` if the plane already exists. The `ID` is written back to the GUI in all the cases and interpreted as shown in the previous paragraph. If the plane already exists in the CA Engine `add_plane()` returns `IGNORED`, and `ID_same` is also written to the GUI. Then immediate visualisation of the plane is enforced by calling `tx_vis_pack()` with its `force` argument set to 1 (see section 8.6.2.1 for more). Finally the CA Engine discards the plane and the function returns `IGNORED`. If the plane did not exist in the CA Engine, it is added to the visualisation list. The function calls `send_ack()` to acknowledge the addition and reads `VIS_PACK` from the socket. It causes immediate visualisation as above and `ADD_PLANE` is returned.

N.B.: The acknowledgement in this case is *not* handled by the calling function `rv()`.

The addition to the visualisation list requires the initialisation of the `cell`. This is achieved by the following function:

```
void init_cell (cell *c_ptr, const plane *pl_ptr, int generation)
```

This sets the forward and backward links of the cell to zero, and assigns the `data` and `generation` members of the cell to those passed to the function as arguments. This function assumes that the memory for the cell to be initialised *has been allocated*.

The `generation` argument is passed equal to the current generation. The cell is then inserted in the visualisation list by means of the function `insert()`.

8.4.5 Why is the Protocol Complicated?

The developers realise that the above protocol is complicated. There are various reasons for this. The `ID_same` token is necessary because the GUI may possibly visualise a plane more

than once, but there is no point in the CA Engine maintaining multiple copies of the same plane.

The immediate visualisation feature was added to the system in response to a specific request from users who wanted to be able to visualise a plane even after the evolution of the automaton had finished [Telford et al. 1999]. Instead of adding another option in the Simulation Window menus we preferred to move the additional complexity to the underlying protocol, which is invisible to the user. The reason why `serv_add_plane()` calls `send_ack()` itself whereas no other function does that, is to ensure that the GUI exits `consume_vis_pack()` (which it always calls when sending requests so as to prevent race conditions). If this is not ensured, the immediate visualisation packet is consumed in the GUI. To this end, `VIS_PACK` had to be added to the protocol as an acknowledgement that `consume_vis_pack()` has been exited.

8.5 Plane Deletion

8.5.1 Deletion Protocol

Plane deletion is initiated by the GUI by sending the ID of the plane to be deleted through the usual `prot_sockfd` socket. The CA Engine deletes the plane with the specified ID from both its lists and acknowledges the deletion. The protocol is shown below:

Sender	Token	Type
GUI	DEL_PLANE	req_code
GUI	ID	int
CA	DEL_PLANE	req_code

8.5.2 The Function `rem_plane()` and Other Related Functions

The function removes the plane with the given ID from the plane list. The prototype of the function is as follows:

```
int rem_plane (plane_list *pl_l_ptr, int ID)
```

Given that the `plane_list` structure is implemented as an array, the plane to be removed is trivially located. A removed plane is signified in the list by a `NULL` pointer. The function checks if the `ID` is legally defined and if the corresponding pointer points to a plane. If this

is not true the function returns `IGNORED`. If the plane is found the pointer is set to `NULL` and the `no_plane` member of the `plane_list` is decremented. `DEL_PLANE` is then returned. Note that the function does *not* deallocate the memory space occupied by the plane.

This is done by the function `disc_plane()`, which, as explained previously in the discussion of the `plane` and `plane_class` functions, also calls `del_pl_class()` to free the only dynamically allocated member of the struct, `class_ptr`.

8.5.3 GUI-Side Plane Deletion

Deletion on the GUI side is handled by the following function:

```
int req_del_plane (int ID)
```

This function writes the `ID` of the plane to be deleted to the GUI, then reads the acknowledgement by means of the `get_ack()` function. If the acknowledgement is `IGNORED`, then `get_ack` returns 0, in which case the function returns 0 as well. Otherwise, the function calls `disc_plane()` to free the memory and `rem_plane()` to remove its entry from the `all_planes` list. These *must be called in that sequence*, because the only handle to the plane is `all_planes.planes[ID]`; if we remove it from the list first, we can no longer access it to free its memory. We then compare the return value of `rem_plane()` with that of `get_ack()`. If they are not the same then there is an inconsistency between the GUI and the CA side and the program exits. Otherwise, `DEL_PLANE` is returned.

8.5.4 CA Engine-Side Plane Deletion

This is handled by the function

```
int serv_del_plane (void)
```

The root process of the CA Engine reads through the socket the `ID` of the plane to be deleted and broadcasts it to the other processes. In addition to what the GUI has to do, the CA Engine must remove the plane from the visualisation list as well.

To do this, it calls the function `del_ID()`. As mentioned when discussing the list-related functions, `del_ID()` returns `DEL_PLANE` if the plane is found; otherwise it returns `IGNORED`. In the former case, `rem_plane()` is called to remove the plane from the plane list and its returned value is returned by `serv_del_plane()`.

8.6 Plane Visualisation

8.6.1 Visualisation Protocol

The visualisation data transmission is initialised by the CA Engine. The GUI, via X, polls the dedicated socket `vis_sockfd` for the code indicating a visualisation packet (`VIS_PACK`), then receives the plane ID and the actual data using `eng_rx_callback()` and `rv_vis_pack()`.

Sender	Token	Type
CA	VIS_PACK	req_code
CA	ID	int
CA	val_size	int
CA	minmax[2]	double[]
CA	[data]	u_char[]

8.6.2 CA Side Visualisation

Suppose that a plane has been added to the visualisation list of the CA Engine. After the CA Engine runs a generation it checks the visualisation list for planes to be visualised in this generation. When it is time for a plane to be visualised, it is popped from the visualisation list. Uniqueness of data transmitted is guaranteed by means of the plane class on the CA Engine side. After the visualisation, the plane is reinserted with its cell's `generation` member altered to match its next visualisation generation.

8.6.2.1 Function `tx_vis_pack()`

The implementation of the visualisation protocol is handled by the function

```
int tx_vis_pack (cell *c_ptr, char force)
```

The function verifies that the plane in the `data` member of the `cell` passed as its argument has not been visualised in the current step. If the `last_vis` member of the plane class of the plane is equal to the current generation *and* `force` is not set, the function returns immediately with `VIS_PACK` as its exit code.

In the general case when the plane is visualised, the processes execute the same steps we have described in section 7.5.3.4, in order to determine which processes are working, as well as the buffer and loop sizes and allocate memory accordingly. In addition, an unsigned character array of size equal to the total extent of the data to be written to the socket (i.e. the number of elements equals the number of cells in the model and the size of each of them is that of an unsigned character) is allocated memory and is used for the colour mapping of the data as described in the next section (8.6.2.2). In order for the fourth item of the protocol, namely `minmax[2]`, to be written, this must be first calculated by traversing the cells which are going to be visualised according to the plane specification seeking the minimum and maximum values for the substate. Traversal is executed in the same way that the local CA copies are traversed for writing data on file. After these limits have been calculated in each process, the results are combined with those of the other processes so as to acquire the global minimum and maximum values for the substate in question. This step is skipped if the user defines the minimum and maximum values manually, as described in section 8.6.2.3. The data are colour-mapped in the processes where they reside before being gathered in process 0, following the same strategy as `serv_view_state()` (see section 7.5.4). The root process writes to the GUI the first four items of the protocol shown in section 8.6.1, followed by the data which are transmitted using one `writen` call.

The function returns `VIS_PACK` on all cases, since all possible errors (failed `write` or `malloc`, for example) are fatal and cause the program to abort.

8.6.2.2 Colour Mapping

As mentioned earlier, the minimum and maximum values for the visualised substates are stored as double precision numbers globally in the processes. Their values are updated every time the substate is visualised and their values are maintained *throughout the life* of the program. By doing this we generally make the mapping consistent for the planes throughout the life of the program and indicate how the substate changes with respect to time. It is worth noting that, because the granule of visualisation is the plane, 3-D models are broken down to planes on the GUI side in order to visualise them. Therefore, in the first step of the visualisation the first plane of the cube visualised possibly sets the minimum and maximum values to something different than the next planes and could be displayed erroneously; in the next visualisation the minimum and maximum values and therefore the colour mapping, are updated, "converging" to the correct values.

The colour mapping is performed in all the processes before gathering the data at the root process so as to write them to the socket. It is done by means of the following function:

```
void colour_map (const u_char *orig_data, u_char *mapped_data,
                 int stateid, int no_data,
                 double gmin, double gmax)
```

The first argument contains the data and the second is an array initialised by the caller function to contain the mapped data. The `stateid` argument of the function is used to define the type of the data in `orig_data` and `no_data` is the number of elements in it. The minimum value of the substate in `orig_data` is mapped to 1 and the maximum is mapped to 255. The intermediate values are linearly projected to the 1-255 interval. This is done using the obvious formula

$$mapped\_data[i] = \begin{cases} 254\dfrac{orig\_data[i] - g\min}{g\max - g\min} + 1, \text{if } g\min \neq g\max; \\ \\ 1, \text{else.} \end{cases}$$

In order to avoid multiple computations, we calculate 254/(*g*max-*g*min) at the beginning of the function; nonetheless we need to compute this every time we call the function, i.e. once for each process *x*-line. The above mapping leaves 0 as the background colour for 3-D visualisations. In the palettes distributed with CAMELot, this corresponds to Black.

8.6.2.3 Manual Minimum and Maximum Definition

As described in [Telford et al. 1999], users of the system requested a facility to set the minimum and maximum values of a substate manually, so as to be able to view a subset of the visualised substate with greater detail.

To achieve this we introduced the character array `auto_map[NumOfStates]`, each element of which indicates if the user has manually set the limits of the corresponding substate. This can be done using the appropriate menu of the Simulation Window. By means of the same menu the user can revert to the automatic calculation of the limits, using the corresponding button.

The protocol request `SET_MINMAX` on the GUI and the CA Engine is handled by the following functions respectively:

```
int req_set_minmax (int substate, double min, double max)
int serv_set_minmax (void)
```

The former writes its arguments to the CA Engine and then calls `get_ack()`, the value of which it returns. The latter reads (on process 0) the data the GUI sends and broadcasts them to the other processes. If the substate is acceptable and the minimum value received is less than the maximum, the appropriate `minmax[][]` elements are updated and that of `auto-map[]` is cancelled. The CA Engine reverts to the automatic mode if the limits read are both equal to zero, in which case the corresponding element of `auto_map[]` is set. It should be noted that setting the limits manually yields performance benefits because the corresponding search taking place in each plane visualisation of the substate as part of `tx_vis_pack()` is skipped.

8.6.3 GUI Side Visualisation

One global visualisation buffer, `viz_buffer`, is initialised by means of the `init_buffer()` function, when `dev_run()` is called. This is done when the user presses the "Run" button and starts the simulation window, and the same buffer is used for all the planes received.

When a packet is received at the `vis_sockfd` socket, X calls `eng_rx_callback()` which in turn calls `eng_rx_packet()` which, if the header is `VIS_PACK` calls `rv_vis_pack()` to read the data. The `plane2win[]` list corresponding to the plane ID of the visualisation packet is then traversed and `viz_render_plane()` called for all windows currently displaying this plane. The "Current Step" field in the Simulation window is then updated with the generation number in the visualisation packet. The last action is also taken when a packet with the `GEN_NO` header is received.

8.6.3.1 Function `rv_vis_pack()`

Another function contained in `guicomms.c` is

```
int rv_vis_pack (req_code request, int *ID_ptr, double *minmax,
                 u_char *value_ptr)
```

This function is called by the GUI when it detects that the visualisation socket contains a message. This message is passed to the function as the `request` argument, and is tested against `GEN_NO`[21] or `VIS_PACK`, the only acceptable values. In the former case it consumes the generation number following the `GEN_NO` `req_code` by placing it in the space pointed

---

[21] This is an addition from release 1.2 onwards to handle the introduction of the `GEN_NO` `req_code`.

by the `ID_ptr` argument and exits. In this case, the return values of the `minmax` and `value_ptr` *by-reference* arguments is undefined. If on the other hand the `request` equals `VIS_PACK`, the `ID` of the plane visualised and the visualisation data are passed in the `ID_ptr`, `minmax` and `value_ptr` arguments respectively. As described in the protocol discussion earlier in this section, the size of the visualised entity is also passed through the socket; this is used as the `nbytes` argument of the `readn()` call issued to read `value_ptr`.

The function returns -1 if the `request` is not `GEN_NO` or `VIS_PACK` or if any of the `readn()` calls issued fail; otherwise it returns `request`.

9.   Performance of the CA Engine

In this section we will discuss the results from benchmarking the CA Engine. We will explain why parallel computing is necessary for COLOMBO and see how well the model scales. We will also assess the impact of the homogeneous systems optimisation, discussed in section 6.2.1. The automatic inactive strip detection optimisation (section 5.6.2) could not be tested using the bioremediation problem, because the model is not deterministic.

9.1 The Benchmark

We decided to benchmark the performance of the CA Engine on as many power-of-two processors as possible. Apart from the scaling curve, this test also gives an idea of the time taken for one processor to carry out the job and can yield a conclusion about the necessity of parallel computers for the task in hand. Because the model is decomposed across the x axis, the x size of the model defines the amount of parallelisation that can be applied.

The scenario we followed did not involve any visualisation or writing to disk, we were only interested in testing the throughput of the program in a productive environment. The system had to read in the initial configuration, and this time was accounted for in all cases. We consider this normal, since state initialisation is inevitable overhead. In all cases we ran 100 iterations starting from the initial configuration provided by UNICAL and CRA. The timings were taken using the built-in timing facility of CAMELot. In the case of multiple processors, and therefore multiple readings, the comparisons were made using the timing results of process 0.

For the benchmark we used the Cray T3E-900 based at EPCC. The system hosts 344 450 MHz processors, each with a peak performance of 900 MFlops. Most of these processors have 128 MBytes of memory or more. It is worth bearing in mind that Cray is a distributed memory machine and that it does not employ virtual memory; therefore the per element total size of the executable and the memory dynamically allocated at run-time cannot exceed the physical memory size of the element.

We used two versions of the bioremediation code for the fluid dynamic layer, provided by UNICAL and CRA. The first one is a 72x72x13 model with 60 states and 29 parameters. The total size of the substates is approximately 32 Mbytes. The total size of the executable, as estimated from the `top` command on a Sun running Solaris 2.6 is 83 MBytes. In this case, 64 processors was the highest power of two that we could use. However, there is no

point in extending the benchmark beyond 16 processors, because then the size of the boundary data is disproportionate to that of the actual data. For example, in the case of 32 processors, the *x*-size of the actual data in most processors will be 2, which equals the *x*-size of the boundary data. The scaling curve was drawn using the homogeneous system optimisation, but we also ran the same benchmarks without employing it, so as to judge its impact.

UNICAL provided another model with dimension 256x128x13. This model allowed to extend the benchmarking to 32 and 64 processors (again 128 would be overkill). However, the model was now too large to be accommodated in 1 processor (just the two CA copies for the 60 states require approximately 410 MBytes of memory).

9.2 Benchmark Results

9.2.1 Scaling Curve

9.2.1.1 Small Model

The timings follow in Table 22. The Sum field contains the time taken for the update function, the boundary replication and the steering. The Total field also includes the time for the initialisation of the system (building of communicators, memory allocation etc), the substate initialisation, the update of the read copy after the application of the transition function etc. A discussion of the timing facility is available from section 3.2.1.1.3.2. Only two decimal places are quoted in the tables. Speedup is the ratio of Total with 1 processor over Total with the number of processors in question.

| Processors	Sum (sec)	Total (sec)	Speedup	Optimum
1	85.86	102.90	1	1
2	42.06	50.88	2.02	2
4	21.44	26.07	3.94	4
8	12.63	15.10	6.81	8
16	8.22	9.59	10.72	14.40

Table 22: Benchmark results for 1-16 processors on the Cray T3E-900

Because the *x* dimension of the model (72) is not divided by 16, the speedup that can be gained ideally is not 16, but $72/\lceil 72/16 \rceil = 72/5 = 14.40$. We used the *Optimum* column in Table 22 to facilitate comparison with the ideal speedup. The scaling curve which yields from Table 22 is shown in Figure 29.
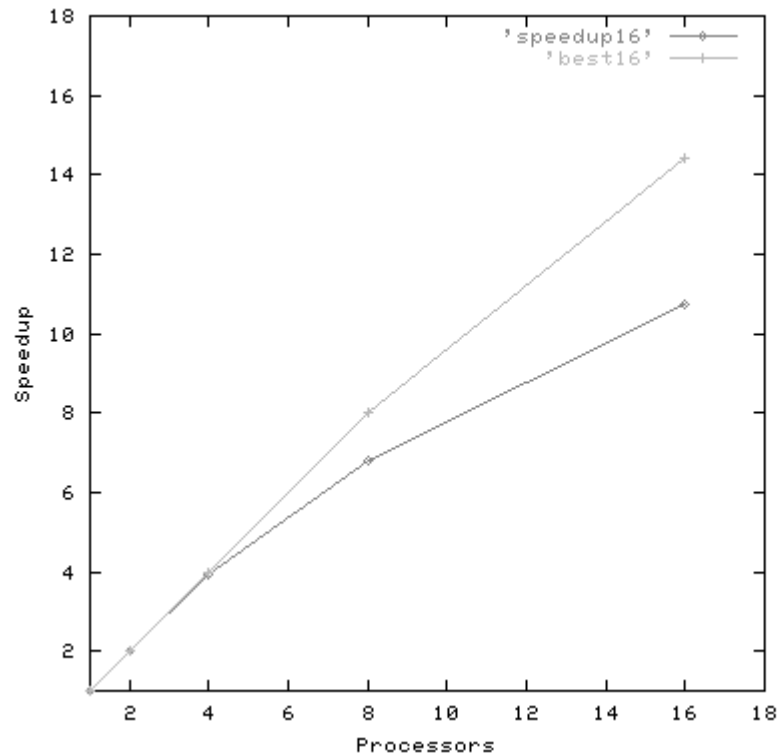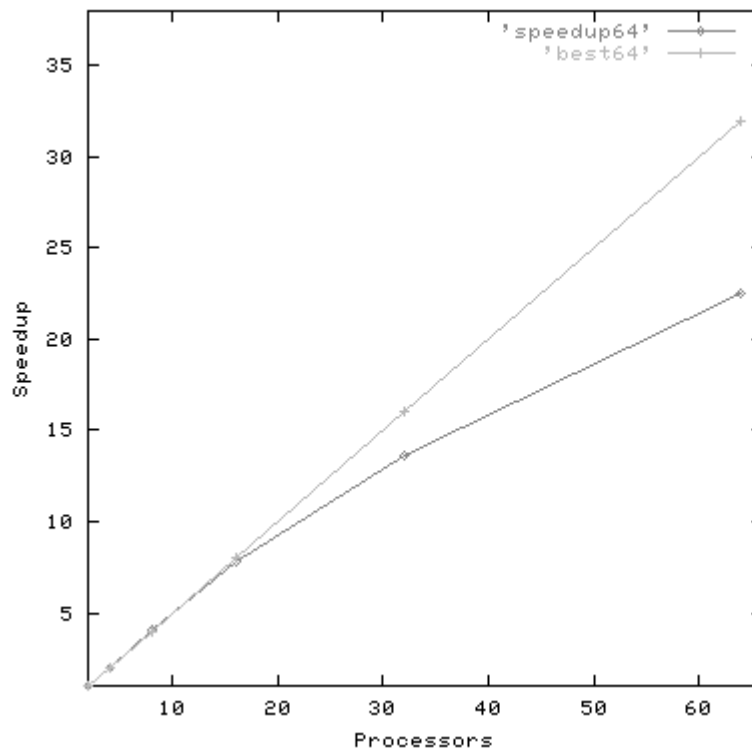

Figure 29: Speedup (red, diamonds) and optimum speedup (green, crosses) scaling curves for the small bioremediation model

9.2.1.2 Large Model

The size of the model caused some difficulties. Apart from the fact that 1 processor could not accommodate the problem, in order to test 2 and 4 processor decomposition it was necessary to employ the large (256 Mbytes) memory elements of the system. The processing element memory size factor was not controlled in the other tests to facilitate scheduling of the batch jobs. Because running the model on 1 processor was not possible, the baseline for the speedup was the performance on 2 processors. The results appear on Table 23 and Figure 30 depicts these timings. Unlike Table 22, the *Optimum* column in this case simply facilitates the comparison between the performance of each case with the 2-processor baseline.

Processors	Sum (sec)	Total (sec)	Speedup	Optimum
2	280.31	332.94	1	1
4	136.31	164.04	2.02	2
8	67.78	81.46	4.08	4
16	35.29	42.56	7.82	8
32	20.42	24.41	13.63	16
64	12.41	14.77	22.54	32

Table 23: Benchmark results of the large model for 2-64 processors on the Cray T3E

Figure 30: Speedup (red, diamonds) and optimum speedup (green, crosses) scaling curves for the large bioremediation model
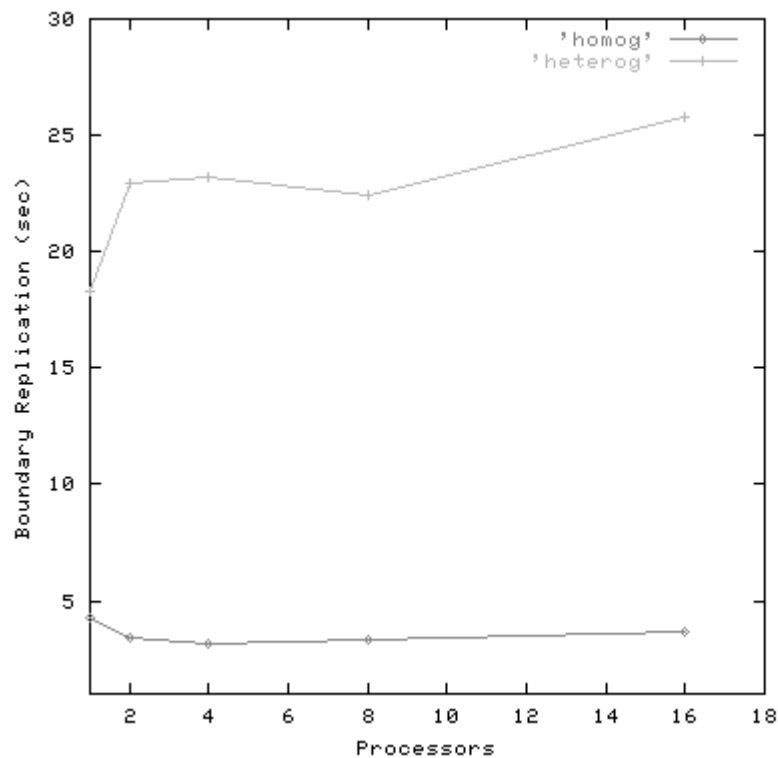
9.2.2 Homogeneous Optimisation

In Table 24 we compare the times taken for the boundary copying with and without enabling the homogenous optimisation for the small benchmark. Figure 31 depicts the results

for the boundary exchange. Similar figures were obtained from the large benchmark and they are not listed as they would not add anything to the discussion.

Processors	Boundary Homog (sec)	Total Homog (sec)	Boundary Heterog (sec)	Total Heterog (sec)
1	4.27	102.90	18.33	115.51
2	3.38	50.88	22.95	70.19
4	3.13	26.07	23.17	46.29
8	3.34	15.10	22.45	34.81
16	3.68	9.59	25.82	30.33

Table 24: Benchmark results for the homogeneous optimisation on 1-16 processors

Figure 31: Graph showing the benefit to the performance of boundary replication when employing the homogeneous optimisation (red, diamonds)

9.2.3 Discussion of the Results

9.2.3.1 Necessity of Parallel Computing

Although CAMELot is a general CA execution platform, the software was developed so as to enable bioremediation modelling. The bioremediation code used as benchmark makes it evident why parallel computing is essential in order to extract modelling results in reasonable amounts of time.

The bioremediation code has two modes. In the first mode, the program runs until it satisfies a set of conditions, called the *equilibrium*. When this happens, the program changes to the second mode where it works directly towards the bioremediation modelling result. This mode is only maintained for one iteration of the CA Engine, and the system then reverts to the first mode seeking the equilibrium conditions. Mode switching is controlled by means of the steering facility.

The number of iterations required in order to reach the equilibrium dominates the running time of the model. This depends on the conditions set and the required accuracy, but in general the first equilibrium takes a lot longer than the subsequent ones. In the past EPCC benchmarked an older version of the bioremediation code. That model was 256x53x5 and consisted of 59 states. The first equilibrium was reached after 225,546 iterations, whereas the next one only needed 1,273 iterations. We attempted to reach equilibrium with the large model discussed in the previous sections. Using 64 processors on the Cray T3E with the homogeneous optimisation enabled, it ran for 12 hours without reaching equilibrium. According to Table 23, this exceeds 290,000 iterations without reaching equilibrium. In such cases the periodic state save facility of CAMELot and its ability to initialise its state from these files are invaluable.

It is therefore evident that parallel computing is essential for realistic modelling of the bioremediation processes.

9.2.3.2 Scaling

The scaling curve in Figure 29 is quite satisfactory. The bioremediation model was only 72 cells long and as a result it could not serve as an ideal benchmark. The 25% difference between the ideal and the actual speedup in the case of 16 processors can be explained by the

fact that the number of actual cells is only 2-2.5 times[22] more that the number of the boundary cells in the macrocell. As it can be seen from Table 24, the boundary exchange accounts for 38% of the total time taken for the model to run. This, and additionally the fact that the curve of the boundary replication time (in the homogeneous case) of Figure 31 is almost flat, indicates that the boundary exchange is the limiting factor. Finally, the seemingly abnormal speedup of 2.02 in the case of 2 processors can be attributed to better caching because the memory size of each macrocell is obviously smaller in this case.

Similar results can be extracted by studying Table 23. The superlinear speedup exhibited in the case of 4 and 8 processors can be attributed to caching again; it would be very interesting to see the results on one processor but this was impossible as mentioned earlier. The boundary exchange is less of an issue in configurations up to 16 or even 32 processors and it seems to affect the speedup drastically on 64 processors. However, when running on 64 processors the size of the boundaries per processor is already half the size of the model portion on the element and still the execution is 65% faster than with 32 processors.

What has been established from these tests is that other the natural bottleneck of the boundary exchange, the CA execution scales well as the number of available processors increases while the size of the per processor data is more than half of the boundary data.

9.2.3.3 Homogeneous Systems Optimisation

This optimisation, discussed in section 6.2.1, has paid off, as it shows on Table 24 and Figure 31. The curve when not enabling the optimisation appears to be rising as the number of processing elements increases. Interestingly enough, the homogeneous optimisation seems to benefit the boundary exchange since the timings appear to be dropping until 8 processors are used although the timing for 16 processors is still less than that for 1 processor. As for the times themselves, the optimisation appears to save from 77% to 86% for the boundary exchange.

---

[22] This is because the decomposition is uneven in this case and some macrocells have *x* dimension 4 and others have 5

10.   Benefits to Project and Open Issues

The benefits that Workpackage 3 provides to other Workpackages of the project are summarised below.

10.1 Benefits to Partners

The software deliverable of Workpackage 3, CAMELot, was used for the development of the bioremediation simulation code, as expected. The partners writing the bioremediation simulation code for the needs of Workpackage 2 helped in the development by providing input on the desired features and bug reports. It is fair to claim that CAMELot is tailored to the bioremediation simulation code, while at the same time it has not attachments to it, remaining a general CA programming and execution tool. CAMELot also contains input from the CABIOR developers who were working on Workpackage 4.

CAMELot is also useful to the partners leading the exploitation Workpackage, 6. The obtained results for the performance of the software were highly satisfactory, offering a good negotiating point in the quest for an exploitation partner. The research publications submitted to various internation conferences also help towards the same goal. Additionally, a press release from EPCC attracted interest from bioremediation companies and the scientific press. The effort is continuing after the end of the project.

10.2 Open Issues

The following issues are possible extensions and optimisations to CAMELot.

10.2.1 Port to Windows NT

As outlined in [Ironside Farrar 1999], most bioremediation companies interviewed would be keen on using the CAMELot software, under the condition that no major modifications or additions to their PC-based computing infrastructure would be necessary. A Linux version of CAMELot is available, however even running Linux is probably not desirable for bioremediation contractors. A more obvious choice would be to run it under an X Window System environment for Microsoft Windows, such as Hummingbird Exceed, but this would incur further performance penalties.

Porting the software to run on Windows NT should be possible, given that X-Designer can produce Windows MFC code and MPI implementations for NT exist. It should be noted that such a port would benefit substantially the market position of CAMELot, as it would make it readily available to its target market.

10.2.2 Single-Processor Optimisation

CAMELot can be used on single-processor systems, although it has been made evident that the usual bioremediation problems are too demanding to run on a single processor system in realistic time. The current implementation employs MPI even in the case of single-processor runs, which incurs an unnecessary performance penalty. A version of CAMELot stripped of MPI-related calls is expected to perform better than the current one in the single-processor case, and should be considered in conjunction with the NT port (section 10.2.1). Extensive modifications are required for this optimisation.

10.2.3 Inactive Strip Detection Enhancements

CAMELot contains an automatic inactive strip detection mechanism, as discussed in section 5.6.2. This mechanism could be enhanced in two ways, discussed below.

10.2.3.1 Automatic Fold Setting

In the current implementation the user must select the number of folds at compile time. It would be useful of they could alter their selection at run time, both interactively and through an appropriate steering statement. This facility would be quite hard to implement. A more important but also more difficult extension, would be to devise an algorithm to set and adjust the number of folds automatically at run-time. This could use the built-in timing facility so as to get information about the performance of the system.

10.2.3.2 Switchable Fold Setting

Currently the user must declare the program as deterministic in order for the inactive fold detection mechanism to take effect (see section 3.3.7). This disables the mechanism in the case of the bioremediation code, because the update function changes after specific events. If a piece of code changes arbitrarily, it is impossible to solve the problem. It is possible however, to enrich the CARPET language with a statement which would denote the start of a deterministic period of execution and another one to end it. Such a modification would render the inactive strip detection mechanism useable in cases like the bioremediation code, when the non-determinism is detectable or caused by the programmer.

10.2.4 Timing Function

As mentioned in section 5.7.1.1, the memory copies at the end of each update are not accounted for in any timer apart from the total one. These should be a part of the update function timer, but it is not straightforward to implement this because the memory copies take place after the boundary copies, which in turn follow the updates. Because the order in which these events happen cannot change, the only way to do this is to extent the interface of the timing functions to include a function which starts adding to a given timer without incrementing the number of calls, and another one to stop this.

10.2.5 Quiescent Substates

In many cases the CARPET programmer may define a set of states which do not change over time at all. A good example of such use could be a substate describing the porosity of the ground in a bioremediation field. This quality is local to each cell and cannot therefore be represented with a global parameter and does not change as the model evolves at any point. Such a state is called *quiescent*.

Currently the CA Engine does not discern between quiescent and normal states. This affects the performance of the system in many ways. The arrays which store the CA data are larger than they could be thus being heavier to communicate in both types of boundary replication and slower to copy in the read copy update after the CA update rule has been applied. Caching of the data to processor memory could also be affected.

This optimisation, suggested quite late in the CAMELot development, requires some modifications to the parser, but the CA Engine code will be very drastically affected. It is however favoured to provide good performance benefits.

10.2.6 Visual cell substate value enquiry

A feature which was requested but could not be implemented within the project timescale was the ability to ascertain the numerical value of a particular cell's substate by selecting the cell visually, using the mouse cursor over a Visualisation window. This would be non-trivial to implement, and would only be useful when the dimensions of the CA are small enough to allow individual cells to be rendered in the Visualisation windows.

11. References

[AVS 1993] *AVS User's Guide*, CST 912, Manchester Computing Centre, University of Manchester, January 1993.

[Baracca et al. 199] *COLOMBO WP4: Functional Requirements and Software Package Design*, M.C. Baracca, P. Ornelli, G. Spezzano, D. Talia, November 1998.

[Booth et al. 1999] *COLOMBO WP3: WP3 Tasks T3.4/3.5 Workplan*, S. Booth, L.Clarke, K. Kavoussanakis, G.Smith, S.Telford, Version 1.1, April 1999.

[Clarke et al. 1998] *COLOMBO WP3: Parallel CA programming Environment*, Deliverable DI3.1.8, L.Clarke, G.Smith, S.Telford, Version 2.0, May 1998.

[Ironside Farrar 1999] *COLOMBO WP6: Scotland/United Kingdom Market Survey*, Ironside Farrar, ref. 5631/MC, October 1999.

[Kavoussanakis et al. 1999] *COLOMBO WP3: Performance of CAMELot 0.2*, Deliverable DI3.2.5, K. Kavoussanakis, S D Telford, S P Booth, Version 1.1, February 1999.

[MPIf 1995] *MPI: A Message-Passing Interface Standard,* Message Passing Interface Forum, Version 1.1, June 1995.

[Spezzano&Talia 1995] *CABOTO WP3: CAMEL Environment User Manual*, Deliverable D5, G.Spezzano, D.Talia, December 1995.

[Smith 1998] *COLOMBO WP3: CABOTO CAMEL Source Code Structure Report*, Deliverable DI3.1.3, G.Smith February 1998.

[Spezzano et al. 1995] *CABOTO WP3: Design and Specification of CAMEL Extension*, Deliverable D2, G.Spezzano, D.Talia, S.Di Gregorio, June 1995.

[Stevens 1990] *UNIX Network Programming*, W. Richard Stevens, Prentice-Hall Software Series, 1990.

[Telford et al. 1998] *COLOMBO WP3: Design for Portable, Parallel CA Software Environment*, Deliverable D6, S.Telford, G.Smith, M.C.Baracca, A.Longo, P.Ornelli, G.Spezzano, D.Talia, May 1998.

[Telford et al. 1999] *COLOMBO WP3: Extensions to CAMELot 1.0*, Deliverable DI3.4.1, S.Telford, K. Kavoussanakis, S Booth, Version 1.1, April 1999.

I. CAMELot Release History

- **1.3** (2000/03/31)

  Internal Software Deliverable **SI3.6.1** (Software Deliverable **S3**). Relaxation of the constraint in the CA Engine on the number of processes, folds and CA x-dimension size; boundary datatype optimisation for homogeneous multiprocessor systems; parameter arrays added to CARPET; revised GUI parameter editor dialog; improved AVS file compatibility, plus changes from 1.2.x releases.

  SunOS 5.6, IRIX 6.2 (N32 ABI), Red Hat Linux 5.2 and Tru64 UNIX 4.0F binary release.

- **1.2.2** (2000/03/15)

  Revised SI3.5.1 release: increased default `yacc` parser stack size to 10000 for Tru64 UNIX, as default size is too small for large CARPET programs.

  Tru64 UNIX 4.0F GUI/parser binary released only.

- **1.2.1** (2000/03/03)

  Revised SI3.5.1 release. Added Tru64 UNIX 4.0F (Alpha) support and changes suggested in COLOMBO WP3 Problem Report 19.

  SunOS 5.6, IRIX 6.2 (N32 ABI), Red Hat Linux 5.2 and Tru64 UNIX 4.0F binary release.

- **1.2** (1999/12/03)

  Internal Software Deliverable **SI3.5.1**. Several bugfixes and optimisations; XDR-format data file support; minor GUI improvements, more CARPET compiler warnings; new `cpt_save()` CARPET steering function; revised C compiler option configuration, plus changes from 1.1.x releases.

  SunOS 5.6, IRIX 6.2 (N32 ABI) and Red Hat Linux 5.2 binary release.

- **1.1.2** (1999/10/20)

Revised SI3.4.2 release: changed user-definable C compiler command line arguments to include `-DCPT_INCLUDE_FILE=` to allow different levels of quote-escaping required for different MPI implementations (i.e. those with an `mpicc` shellscript and those without).

IRIX 6.2 GUI/parser binary released only.

- **1.1.1** (1999/10/07)
  Revised SI3.4.2 release: parser bug fix for problem with incorrect array indexing when using region reduction functions with array substates in CARPET programs.

Red Hat Linux 5.2 binary release (with Metro Link Motif 2.1) only.

- **1.1** (1999/06/10)
  Internal Software Deliverable **SI3.4.2**. Many changes; see Report DI3.4.1. Bugfixes: "Parameter" dialog box now gives correct current parameter value, "Edit Substate" no longer crashes CA Engine. Major efficiency improvements in CA Engine.

  SunOS 5.5.1, IRIX 6.2 and Linux binary release.

- **1.0.1a** (1999/06/08)
  Revised SI3.3.4 release: Release 1.0.1 with parser recompiled due to buggy version of `yacc` being used to build Linux CAMELot 1.0.1.

Linux GUI/parser binary released only.

- **1.0.1** (1999/06/07)
  Revised SI3.3.4 release: parser fix to handle greater numbers of substates, neighbourhoods and parameters, and to detect when the limits on these are exceeded.

SunOS 5.5.1 and Linux GUI/parser binaries released only.

- **1.0** (1999/03/08)

Internal Software Deliverable **SI3.3.4** (Software Deliverable **S2**). Batch mode added to CA Engine; bugfixes to CA Engine; memory leaks fixed; increased CA Engine startup timeout to 20s; optimised visualisation rendering.

SunOS 5.5.1, IRIX 5.3 and Linux binary release.

- **0.2.1** (1999/02/16)
  Revised SI3.2.4 release: parser bug fix to enable `cell_<substate>` access globally (CAMEL CARPET compatibility); added `DimX`, `DimY`, `DimZ`, `NProcs`, `NFolds` constants to CARPET; `cpt_thresh` handling and random function bug fixes.

  SunOS 5.5.1, IRIX 5.3 and Linux binary release.

- **0.2** (1998/12/09)
  Internal Software Deliverable **SI3.2.4** (Software Deliverable **S1**). Added 3-plane isometric visualisation functionality, runtime CA Engine fold control and colour map bar display. Many bugfixes and optimisations.

  SunOS 5.5.1, IRIX 5.3 and Linux binary release.

- **0.1** (1998/10/16)
  Internal Software Deliverable **SI3.2.3**. Added runtime CA Engine control, visualisation functionality and CA folds.

  SunOS 5.5.1 and IRIX 5.3 binary release.

- **0.0.1** (1998/08/04)
  Revised SI3.2.2 release: Changed "MPI arguments" configuration option to "MPI run command" - this now allows more of the command line to be specified. Slight changes to font identifiers needed for IRIX X servers.

  SunOS 5.5.1 and IRIX 5.3 binary release.

- **0.0** (1998/06/18)
  First release, corresponding to Internal Software Deliverable **SI3.2.2**.

SunOS 5.5.1 and IRIX 5.3 binary release.

II. CAMELot MPI Configuration

CAMELot 1.2 and later releases can be configured for various different implementations of MPI using the "C compiler command line " and "MPI run command" dialog boxes. Implementations it has been successfully tested with are listed below:

MPICH 1.1

This is the MPI implementation that CAMELot is configured for by default. It is assumed that the environment variable `$MPIR_ROOT` is set to the root directory of the appropriate MPICH installation.

MPICH 1.2

This requires the following change to the default settings:
C compiler flags: change `-DCPT_INCLUDE_FILE=\\\"%s\\\"` to `-DCPT_INCLUDE_FILE=\"%s\"`.

LAM 6.3

It is assumed that the environment variable `$LAMHOME` is set to the root directory of the appropriate LAM installation. The following change to the default settings are also required:
C compiler name: change `$MPIR_ROOT` to `$LAMHOME`.
C compiler flags: change `-DCPT_INCLUDE_FILE=\\\"%s\\\"` to `-DCPT_INCLUDE_FILE=\"%s\"`.
MPI run command: change `$MPIR_ROOT` to `$LAMHOME`.

SGI MPT 1.3 (IRIX 6)

This requires the following changes to the default settings:
C compiler name: set to `cc`.
C compiler flags: change `-DCPT_INCLUDE_FILE=\\\"%s\\\"` to `-DCPT_INCLUDE_FILE=\"%s\"` and append `-n32` if using IRIX 6.2 or earlier.
C libraries: append `-lmpi`.
MPI run command: change to `mpirun -np %d`.

Sun HPC ClusterTools 3.0

This requires the following changes to the default settings:
C compiler name: set to `tmcc`.
C libraries: append `-lmpi -lnsl`.
MPI run command: depends on HPC ClusterTools environment (CRE or LSF).