The Formal Specification in **Z** of Task Migration on the Testbed Multicomputer

Paul Martin

ECS-CSG-2-94
Department of Computer Science
Edinburgh University
Edinburgh EH9 3JZ, Scotland

June 1, 1994

Abstract

This report introduces a message-passing multicomputer called the 'Testbed' and describes its facilities for transparent task migration between processors. A specification in the formal language **Z** is given for the key operating system components which support migration. Finally, rigorous arguments are presented which verify that task migration is correct and safe. The contributions of this report are an extended specification showing the application of **Z** to a real system and a detailed demonstration of the verification of safety and correctness properties. Conclusions are drawn about the utility of formal methods in general.

Contents

1 Introduction

This report introduces an experimental message-passing multicomputer called the 'Testbed' which was developed at Edinburgh University. The Testbed supports the migration of user tasks between processors for the purposes of load balancing. A key property of the migration is that it is automatic and transparent, i.e. migration occurs without user intervention and does not affect the final outcome of the computation.

The protocols necessary to control and coordinate the migration and sharing between processors of objects such as user tasks, inter-task communication channels and user memory areas are complicated. This complication arises from the strict requirement that migration be transparent and the practical requirement that the migration overheads be as small as possible. The use of formal methods was deemed to be the best way to cope with the complexity of the protocols.

Formal specification is beneficial to the protocol designer in two ways. Firstly, the creation of the specification itself forces the designer to consider the protocols in some depth, both in order to select the correct level of abstraction and, later on, in order to ensure that the specification is complete. Secondly, once the initial drafts of the specification have been produced the verification phase can begin. Verification forces the designer to state precisely the properties that the protocols must exhibit and to state any assumptions which the protocols make about their environment. A good deal of interaction between the verification and specification phases is to be expected.

The report is organised as follows. In the next section the Testbed computer is introduced in terms of its hardware and software components. This provides the reader with an understanding of the real system which is modelled in the specification. In Section 3 a brief description of the **Z** language is presented. Section 4 contains the bulk of the specification, formally defining the Testbed protocols for task synchronisation and inter-task communication. A short Section 5 builds on this work and specifies the task migration protocols. The rigorous proofs which verify the safety and correctness of the task migration follow in Section 6. The final section concludes the report by discussing the utility of formal methods.

2 The Testbed Multicomputer

This section introduces the Testbed and is in two parts: a description of the conventional aspects of the Testbed is followed by a description of the special operating system functions which support task migration.

2.1 Conventional Features

The Testbed (also described in Imre [6]) is an experimental, distributed memory, message-passing multicomputer constructed at the University of Edinburgh

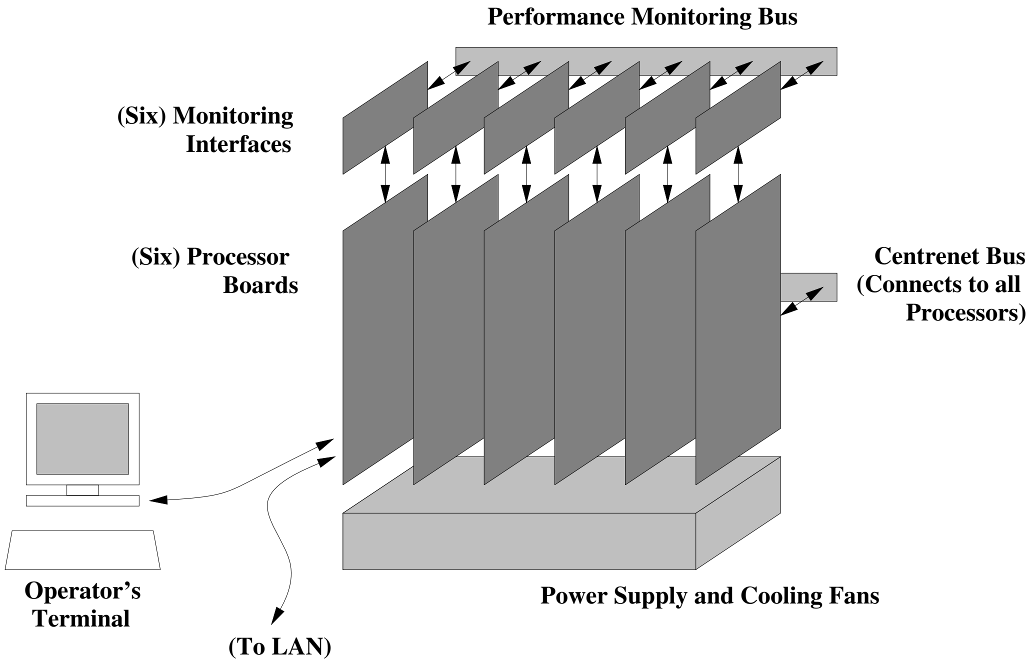

Figure 1: *An overview of the Testbed hardware.*

between 1988 and 1991. A free-standing cabinet about five feet high contains a power supply, cooling fans, six processor boards, a bus-based processor interconnect called Centrenet (a detailed description of which can be found in Ibbett *et al* [5]) and special hardware for dynamic performance monitoring. A single-user machine, the Testbed has one RS-232 link to a terminal for operator control and a second serial link to the local area network and hence access to a filestore. Figure 1 gives an overview of the Testbed hardware.

The Testbed operating system (TOS) is written in C and provides a time-sliced, multi-tasking environment. TOS has a built-in shell which offers a typical Unix interface to the user. TOS is replicated over all processor boards and the console may be switched (in software) to communicate with any of the six shells. Tasks may be invoked on any operating system and migrated between operating systems transparently to the user. Figure 2 gives an overview of the Testbed system software.

Some small utility programs have been implemented on the Testbed offering similar features to the Unix programs `more`, `grep`, `compress` and `wc`. A disassembler and a version of the editor `ue` have also been ported. However, as no compiler has been implemented, all new programs must be cross-compiled on another machine and then up-loaded to the Testbed via the LAN connection. A typical experimental session might be as follows: edit and compile the test program on a Unix workstation; up-load the binary to the Testbed; execute the program on the Testbed collecting the results in a file; down-load the file to the workstation; and analyse and display the results with, for example, `perl` and `gnuplot`.

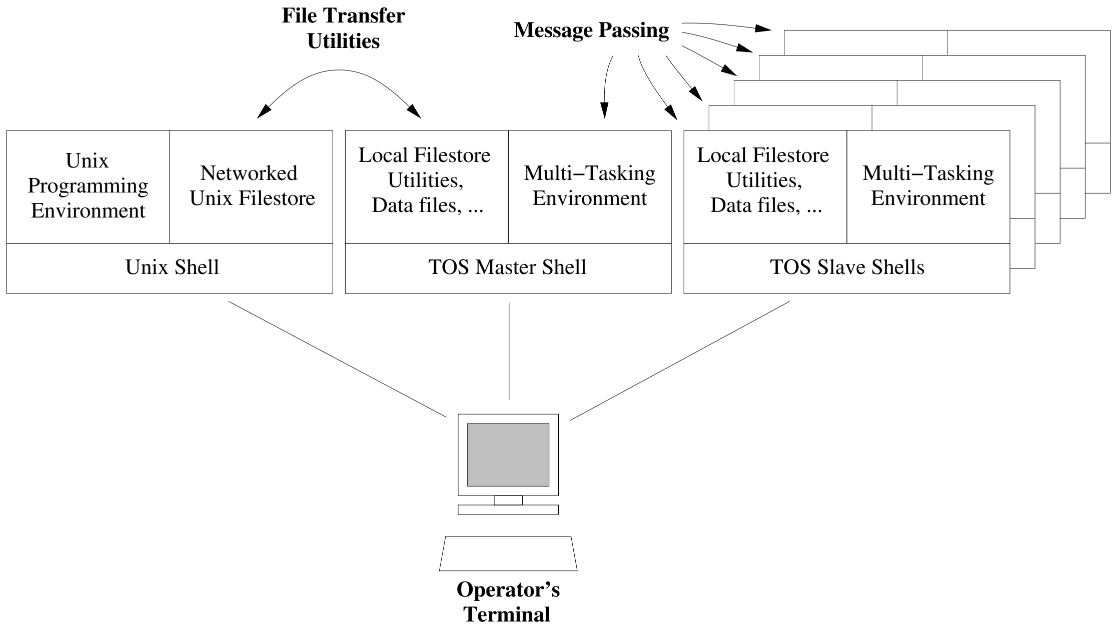

Figure 2: *An overview of the Testbed software.*

2.1.1 The parallel processor boards

Each processor board has a Motorola 68010 processor, several megabytes of RAM, support for virtual memory, a Centrenet controller (including a Direct Memory Access capability) and a connection to the performance monitoring bus. The board designated as 'master' also has drivers for two serial links and *reads* event data from the performance monitoring bus—the other, 'slave' boards may only *write* to the bus. The performance monitoring bus is used by the master processor to gather information about the current load as a basis for controlling task migration.

The number of processors and their 1.2 MIPS performance put the Testbed in the MIMD, medium granularity class, somewhere between the Cray-XMP approach, which uses a small number of very high performance processors, and the DAP approach, which uses a large number of simple processors.

The Testbed is designed as a message passing architecture rather than a shared memory architecture for several reasons. Firstly, the occam model of programming supported by the Testbed explicitly communicates sequences of bytes over channels and only shares variables if they are read but not written. Secondly, the interconnection network required for message passing is generally simpler to implement (and more scalable) than that required for shared memory since global memory updates are avoided.

The Centrenet interconnection network has a hierarchical design: nodes, which comprise up to sixteen processors sharing a bus, are connected by fibre optic cable into a tree. Communication time is minimal if the source and destination processors are part of the same node; otherwise the communication time is proportional to the number of nodes traversed. Since the Testbed has just six processors, its communication network requires only a single Centrenet node. The communication speed is approximately 10Mbytes/second.

2.1.2 The Testbed operating system

Each replication of TOS on each processor may host up to sixteen processes at any one time. In this context, a 'process' is created every time a new program is invoked by the user. Each process has a code and a data segment of up to half a megabyte. Strict rules prevent one process from modifying the data of another process, although multiple invocations of the same program may share a code segment and trusted programs (such as debuggers) may have read-only access to other processes' memory. Processes may be marked as 'foreground', 'background' or 'suspended' (for debugging purposes). The command `ps` lists the processes, their 'threads' (described next) and optionally the thread contexts, `kill` may be used to remove processes and the commands `fg` and `bg` move processes between the foreground and background.

In TOS terminology, it is not processes which execute but 'threads'. Each thread has its own program counter and stack but shares code and global data with the other threads in the same process. Each process may own up to sixty-four threads, the threads being multi-tasked on an equal priority, round-robin basis.

Threads have a maximum time-slice of 20ms, although they may be pre-empted if they call certain operating system services. TOS has been made pre-emptive so that interactive programs will operate correctly, but the time-slice has been set relatively high in order to reduce the number of context switches per unit time. This is necessary because the context switch takes a long time, as much as fifty times as long as a Transputer context switch. Threads communicate over occam-style channels (as described in Subsection 2.1.3) and each process may own up to 128 channels.

The Testbed does not have backing store on which to keep parts of the virtual memory that have been 'paged out' so Testbed programs must be conservative in their use of memory. Practical experience, however, shows that the available RAM is almost always sufficient.

Each replication of TOS maintains its own filestore in local memory. The filestore is effectively a one-level directory and holds program files, scripts, data files and configuration files. Up to sixty-four files are allowed, a maximum size of half a megabyte per file being imposed. Files may be copied between processor boards by the user (versions of the Unix commands `ls`, `cp`, `rm` and `mv` are available) and executables are automatically mounted as needed. File permissions may be set with `chmod` to specify execute, read or write.

Other assorted features include a real-time clock, simple script interpretation, a form of environment variables, redirection of `stdout` and some terminal control via `stty`.

2.1.3 Programming environment

Programs to be executed on the Testbed are written in C, compiled and linked with Testbed-specific libraries. Most of these libraries are implementations of the standard C library functions described in Kernighan and Ritchie [7, Appendix B] and the rest are new extensions to the C programming language to allow control of multiple threads and channel communication. Here is a summary of the standard functions available on the Testbed.

- The `stdio` functions for opening, flushing, closing, seeking and unlinking files, formatted printing (`fprintf`), character reading and writing, block reading and writing.
- The `string` functions for copying, concatenating, comparing and searching strings and memory block copying.
- Some of the `stdlib` functions for string-to-integer conversion, memory allocation, `exit` and environment variable search operations.
- Some of the `time` functions for reading the real time clock.

The `ctype`, `assert`, `stdarg`, `setjmp`, `signal`, `limits`, `float` and `math` functions have not been implemented.

The thread model The first area in which new extensions to the C programming language have been made is that of thread control. Thread control is based on the simple yet powerful process model of `occam`, as embodied by the `PAR` statement: a parent spawns a number of children and is blocked until the children complete. Details of the `occam` language can be found in Pountain [13] and INMOS [8] and a discussion of the unique benefits of `occam` in Welch [16]. Thread control is an operating system function, accessed by means of the following library routines:

`int create(id, n_pages, processor, entry, stack)` A child thread with ID number derived from `id` is created; the child is allocated `n_pages` of stack space; its initial program counter and stack pointer are loaded from `entry` and `stack`; the child is queued for execution on the processor board specified by `processor` or chosen randomly if `processor` has a negative value. The value returned by `create` is zero if the child cannot be created, otherwise the ID number of the child.

`wait` Once the parent has called `create` for each new child, it calls the `wait` function and is suspended until its children have terminated.

`exit(err)` Threads terminate by calling the `exit` routine and passing an error code. Two of these codes are reserved for the `occam` `HALT` and `STOP` conditions.

`occam` is a static language in terms of process creation and channel communication. In `occam` the identity of all processes and channels can be known at compile time. This has the advantage for architectures without large virtual address spaces (such as the Transputer) that all memory requirements are known before the program is executed and memory can be allocated statically. Fully dynamic process creation is possible on the Testbed but passing the thread `id` value is more in keeping with the `occam` philosophy and simplifies the use of debugging programs, such as that designed by Woods [18].

The communication model The second area in which new extensions to the C programming language have been made is that of inter-thread communication. The model of communication is based on `occam`: a pair of threads wishing to communicate reserve (for the entire program execution) a unique, unidirectional channel. One thread performs send operations on the channel, the other receive operations. Both threads are blocked while communication completes.

Communication on the Testbed is implemented in the operating system and accessed through the following library routines:

`send_block(chan, buffer, length)` A thread requests to send a message of `length` bytes beginning at the address given by `buffer` on channel `chan`.

`int receive(chan, buffer, length)` A thread requests to receive at most `length` bytes of data beginning at the address given by `buffer` on channel `chan`. The return value specifies the actual number of bytes received.

2.2 Support for Task Migration

In addition to the functionality of conventional parallel computers, the Testbed offers task migration. This advanced feature require special operating system data structures for representing tasks and protocols for migrating tasks and their resources. The extensions made in both of these areas are described below.

2.2.1 Special data structures

The unit of migration on the Testbed is the thread. To make the migration of a thread t from source processor sp to destination processor dp as efficient as possible it must be easy to 'disconnect' the data structure that represents t from its environment at sp, pack t into a message and transmit the message to dp. At dp, the reverse process of unpacking and reconnecting t must also be made simple. Migration is complicated by the fact that threads use local resources, such as communication channels and pages of memory, and for transparent thread migration these must be moved or copied in a consistent way.

Experience with the Testbed shows that in addition to the usual considerations when designing data structures—i.e. minimal size, simplicity for ease of imple-

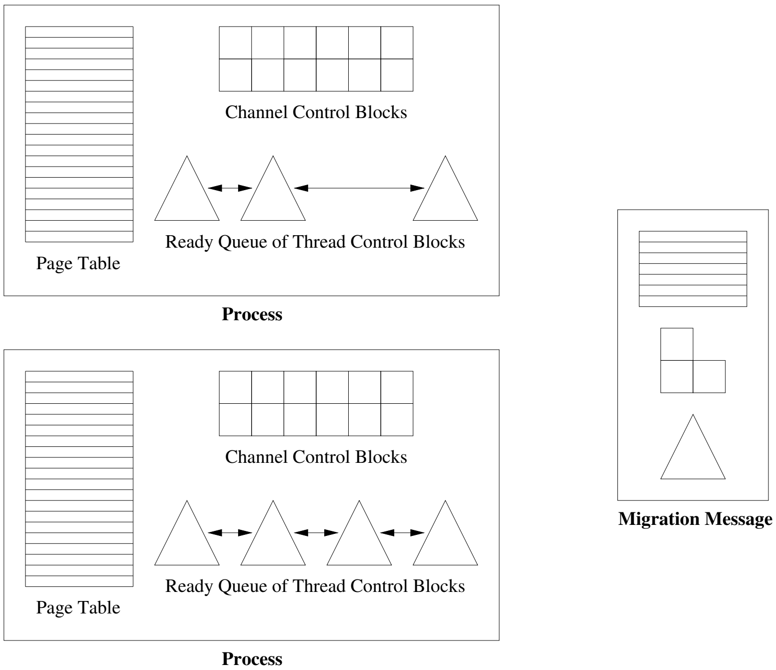

Figure 3: *A processor with two executing processes and a migrating thread waiting for transmission.*

mentation and maintenance, and efficiency of access—the data structures used in TOS to represent threads and their resources must have the following properties.

1. All the relevant data structures must be kept near each other, both to assist allocation and release, and in order that they may be located speedily during migration.

2. The number of dependencies or links between data structures must be minimised to speed and simplify 'disconnection' and 'reconnection'. Data structures such as doubly-linked lists are necessary.

In TOS, a single record called the Thread Control Block (TCB) is used to represent a thread. Unused TCBs are stored on a free list and allocated when a thread is created or arrives during migration from another processor. TCBs are destroyed when a thread terminates or when it is migrated away. When a thread is to be migrated, the information to be transmitted is localised in three areas: the TCB which holds register context and other control values; the Channel Control Blocks (CCBs) which store the status of channels used by the thread; and the virtual memory page table which stores information on memory pages owned or shared by the thread. The 'disconnection' of a thread simply requires that it be unlinked from the process ready queue.

An example migration is illustrated in Figure 3. On one of the Testbed processors two processes are executing. Each process has a virtual memory page table, an array of CCBs and a doubly-linked list of ready threads. In addition, a message containing a migrating thread awaits transmission. This message contains the TCB and copies of the relevant CCBs and memory page table entries.

2.2.2 Migration protocols

The migration protocols define the rules for moving and copying threads, memory pages and channel information between processors. These rules are needed to ensure migration transparency and to prevent, for example, the creation of multiple copies of a thread occurring, updates to the same memory page at different sites happening or the loss of messages on channels used by a migrating thread.

The migration protocols are complicated, principally because they have to deal with concurrent interactions between multiple processors, and their design was greatly assisted by the development of the formal specification. In fact, several major alterations to the initial implementation were made when the specification showed that problems might arise in certain unusual sets of circumstances.

Any thread may be migrated as many times as desired, it is possible for several migrations to occur at the same time and no destination processor may refuse to accept a migrating thread. However, a source processor can refuse to send a thread if the thread is in the wrong state. Threads may be in only one state at a time and examples of possible states are: in the ready queue, waiting to communicate, waiting for a page of virtual memory or, indeed, in the act of being migrated. The TOS protocol states that only threads currently in the ready queue may be migrated. Without this restriction, different 'disconnection' (and 'reconnection') procedures would be needed for threads in each state. The motivation for distinguishing the ready state is that ready threads are the most likely to consume valuable system resources in the near future, and hence are likely to be good candidates for migration.

The protocol for moving and copying pages of memory between processors is too complicated to state in full here, although the following demands are placed upon the protocol. For efficiency reasons, pages from the read-only code segment of a process may be freely copied around the system whenever a migrating thread requires them. Following the semantics of occam, if a parent thread pt creates a child thread ct then ct may read (but not modify) variables in pt's stack area—such pages are copied if necessary. Pages holding the stack area of a thread that has just migrated may be copied if the thread uses them again. Finally, the memory page protocol also has to know when to flush out-of-date copies of pages.

The third protocol, for communication, is the most complicated of all. For efficiency reasons, CCBs are distributed and changes due to thread migration are not made globally—much care was taken in protocol design to ensure that all copies of a process' CCBs stay in a consistent state. Without presenting the complete protocol here, the following remarks are made. Suppose that threads t_1 and t_2 communicate over channel c. When they are on the same processor then a single CCB is used to represent c. When they are on different processors then two

CCBs are needed. If t_1 migrates away from t_2 then information must be extracted from the shared CCB and used to create a new CCB. If t_1 migrates onto the same processor as t_2 then the information from two CCBs needs to be combined in a shared CCB.

3 A Short Introduction to the Z Language

In this section I introduce enough **Z** notation and semantics for readers unfamiliar with the language but with some mathematical background to gain an understanding of the formal specification in Sections 4 and 5. The specification style that I have used is fairly restricted, so it is by no means necessary to understand the whole of **Z** in order to understand this report.

Z is a formal specification language developed by Oxford University's Programming Research Group. **Z** is based on first-order logic and set theory and this makes it possible to express mathematical proofs in the language. Objects at the disposal of the **Z** specifier include sets, relations, functions, sequences and bags. Three good introductory articles on specification are: Meyer [11] in which a charming illustration is given of why formal methods are preferred to natural language for specification; Hall [2] in which some common myths about specifications are exploded; and Wing [17] in which the range of available formal methods is surveyed. A detailed description of the **Z** language can be found in the reference manual by Spivey [15] and a good tutorials in Diller [1].

3.1 Schemas

A **Z** specification is primarily a collection of schemas. Each schema is a grouping of declarations and predicates chosen to represent some part or aspect of the system being modelled. Declarations introduce new variables and assign types—variables must be declared before use and declarations are global. Predicates express relationships between variables. Schemas are generally used either to express the state associated with the system being modelled or operations on that state. A typical state schema contains a list of variables and some predicates expressing constraints or invariants on the state. A typical operation schema includes a state schema and, using a method for distinguishing the 'before' and 'after' state variables, the predicates show how the operation updates the state.

Consider the two example schemas *DBState* and *AddMike* which might be part of a specification modelling a database of employees and their phone numbers.

DBState

$employees : \mathbb{P}\,PERSON$
$extensions : PERSON \leftrightarrow \mathbb{N}$

$\mathrm{dom}\ extensions \subseteq employees$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddMike}} \\
\hline
\Delta DBState \\
\hline
employees' = employees \cup \{Mike\} \\
extensions' = extensions \cup \{Mike \mapsto 1234\} \\
\hline
\end{array}
$$

The schema boxes group the declarations (above the horizontal line) and the predicates (below the horizontal line). The first schema specifies how the state of the database is represented: the *employees* variable is a set of persons; the *extensions* variable is a set of ordered pairs where each pair contains a person and a number. The second schema specifies the state-changing operation of adding a new employee Mike with extension number 1234 to the database.

The $\Delta DBState$ schema declares the variables of *DBState*, *employees* and *extensions* and (because of the 'Δ') also the primed (or 'decorated') variables $employees'$ and $extensions'$. These variables obey the relevant predicate. It is the convention that an undecorated variable represents a component of the state *before* the operation and a decorated variable represents a component of the state *after* the operation.

It is also the convention that an operation schema should define the value of every decorated state component declared. For reasons of clarity and parsimony the specification in this report deviates from this convention and readers should assume that *where the value of a decorated state component is not defined it retains the value of the corresponding undecorated state component.* A formal treatment of this issue can be found in Pitt and Byers [12].

Multiple predicates in a schema are implicitly conjoined. Identifiers in the predicates must be declared in the upper part of the schema or must be declared as global variables. When a schema name $S2$ appears as a declaration in another schema $S1$, the declarations of $S2$ are merged with those of $S1$ and the predicates of $S2$ are conjoined with the predicates of $S1$. When a schema name $S1$ appears as part of a predicate P, the predicate part of $S1$ may be substituted into P.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddPerson}} \\
\hline
\Delta DBState \\
p? : PERSON \\
e? : \mathbb{N} \\
\hline
employees' = employees \cup \{p?\} \\
extensions' = extensions \cup \{p? \mapsto e?\} \\
\hline
\end{array}
$$

The *AddPerson* schema illustrates another feature of schemas: input and output arguments. The $p?$ and $e?$ variables are not part of the state—they are input variables to the operation. Output variables defined by the operation are also possible—identifiers have an exclamation point appended.

3.2 Sets

Here is a summary of the **Z** notation for various sets and operations on sets:

Enumeration: The ::= notation is used when specifying a set by enumeration, e.g. $INSTRUCTION ::= send \mid receive \mid stop$. (When a set is to be used as a new type, it is usually given an identifier in upper case.)

Comprehension: Set comprehension uses the $\{signature \mid predicate \bullet term\}$ notation, e.g. $\{n : \mathbb{N} \mid n \neq 0 \land n \bmod 2 = 0 \bullet n\}$ specifies the set of positive, even numbers as does $\{n : \mathbb{N} \mid n \neq 0 \bullet 2 * n\}$.

Equality: The $=$ symbol is used to express equality as in $\{1, 2, 3\} = \{3, 2, 1\}$.

Empty set: The symbol $\varnothing$ is the preferred way to denote the empty set.

Power set: The declaration $x : \mathbb{N}$ declares x to be a natural number but the declaration $X : \mathbb{P}\,\mathbb{N}$ declares X to be a set of natural numbers.

Operators: Set difference is denoted by $\setminus$, domain anti-restriction by $⩤$, e.g. $\{a\} ⩤ \{a \mapsto 1, a \mapsto 2, b \mapsto 2, b \mapsto 3\} = \{b \mapsto 2, b \mapsto 3\}$, and function overriding by $\oplus$.

Domains and ranges: The dom operation takes a relation and delivers the domain, ran delivers the range.

Sequences: The sequence is just a particular kind of function. There are some predefined operations on sequences: *head* yields the first element in a sequence, *tail* everything but the head, and $\frown$ joins sequences. When a sequence is listed it should appear between sequence brackets $\langle\rangle$.

3.3 Relations and functions

Relations and functions are a convenient way of associating different bits of data with each other:

Relations: If R is a relation between sets X and Y then R is a subset of the Cartesian product $X \times Y$ and this is denoted in **Z** by $R : X \leftrightarrow Y$. If, for example, the ordered pair (x, y) is a member of R then the **Z** notation $x \mapsto y \in R$ can be used.

Total functions: If F is a total function from X to Y then $F : X \rightarrow Y$ is written meaning every member of X maps to (exactly) one member of Y.

Partial functions: If F is a partial function from X to Y $F : X \nrightarrow Y$ is written to mean that a member of X either does not map at all or maps to exactly one member of Y.

3.4 Miscellaneous Z notation

Abbreviated definition: The == symbol is used to define the left hand side as an abbreviation for the right hand side, e.g. $small\_evens == \{2, 4, 6, 8\}$.

Defining types: **Z** has a few built-in types ($\mathbb{N}$, the set of natural numbers, is the only one of these used in the specification), the ability to define new types based on existing types, by using $\times$ or $\rightarrow$ for example, and the ability to define types without saying exactly what they are. For example, the notation $[PERSON, PHONE]$ defines two 'given types' representing, presumably, people and telephones.

Defining operators: **Z** contains many operators for manipulating relations, sequences, and other kinds of sets but it also has the ability to define new operators using 'axiomatic definitions', e.g.

$$\begin{array}{|l}
square : \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall\, n : \mathbb{N} \bullet square(n) = n * n
\end{array}$$

Quantification: An example of existential quantification is $\exists\, x : \mathbb{N} \mid x \leq 5 \bullet x = x * x$ meaning that there exists a natural number x (which has the property of being less than 5) such that x is its own square. An example of universal quantification is $\forall\, x : \mathbb{N} \bullet x \geq x$.

Tuples: An example pair could be written $(1, x \mapsto y)$ and the first member of the tuple can be extracted with $first(1, x \mapsto y) = 1$.

3.5 Checking the specification

The **Z** specification presented in this report has been checked in six different ways.

1. The *f*UZZ software package by Spivey [14] has been used to check that the specification complies with the **Z** rules for syntax, typing and scoping.

2. Simple indexing tools were used to help with locating declarations and dependencies.

3. Informal walk-throughs of the specification and the accompanying documentation helped to identify areas in which the model being proposed was inconsistent.

4. Informal comparison of the specification with the implementation code was used to justify the claim that the specification models the implementation.

5. The formulation and proof of statements about the specification enhanced confidence.

6. Executing the implementation (real-world testing) gave some confidence that gross errors had been eliminated, although the effort required to test under all possible conditions was prohibitively expensive.

It is worth noting that the use of *f*UZZ, which requires declaration-before-use, in conjunction with the lack of modularity in **Z** (when compared, for instance, to VDM as in Hayes [3]) strongly influences the order of presentation in Sections 4 and 5 and produces a distinctly bottom-up style.

4 Specification of Testbed Basics

The main part of this section is concerned with the specification of the task control and channel communication modules of the Testbed's operating system (TOS). The following section contains the specification of the migration protocols. The latter part of this section explains in detail how the **Z** specification schemas are related to C functions in the implementation.

The specification was carried out with three aims in mind. Firstly, it should explain (at a suitable level of abstraction) the inner workings of the operating system. Secondly, it should form the basis for the proofs of correctness. Thirdly, it should help deal with the enormous complexity of adding task migration to a multicomputer. Specification is often used for refinement purposes but this, unfortunately, is outside the scope of the report.

The specification does not attempt to model all parts of the operating system as this would require a considerable amount of time to complete. Instead, it concentrates on the functions related to tread synchronisation, channel communication and thread migration because these are the most difficult to implement correctly.

4.1 Thread Design

I begin this section by introducing the basic objects in the specification such as threads, processors, counters and the message types exchanged between the processors. I define that part of the Testbed state to do with thread control in terms of the relationships between parent and child threads, threads and counters, processors and threads and so on. The state changing schemas are introduced, first at a lower level showing the mechanisms for the task control and then at a higher level showing in what circumstances each state change may occur.

Appendix A contains an index of all globally declared schemas, relations and types used in the specification.

4.1.1 Basic objects

I declare the existence of two given sets, $THREAD$ and PE, which the type-checker interprets as user-defined types and the reader should interpret as the set of all threads that execute on the Testbed and the set of Testbed processing elements (processors).

$[THREAD, PE]$

Later on in the specification I will want to be able to use the idea of 'no thread' and to do this I distinguish a member of the set $THREAD$ by naming it $null\_thread$.

$$null\_thread : THREAD$$

I now model TOS 'counters' which are used to implement thread synchronisation. Parent threads have a counter associated with them which is incremented every time they create a child and decremented every time one of their children terminates. The parent can request to be suspended on the completion of its children, i.e. it can ask to be blocked until its counter has reached zero. The following piece of notation declares $COUNTER$ to be an abbreviation for the set of pairs of natural numbers and threads.

$$COUNTER == \mathbb{N} \times THREAD$$

The natural number part of variables of the $COUNTER$ type will be used to represent the number of children not yet terminated and the $THREAD$ part will be used to represent the parent thread blocked on the counter—where no parent is blocked the value $null\_thread$ will be used instead.

The Testbed processors synchronise their actions by communicating various kinds of control message over Centrenet. The type $CNETMSG$ is defined for these messages.

$$\begin{aligned} CNETMSG ::= {} & focusm\langle\langle PE \times CHANNEL \times PE \rangle\rangle \\ & \mid\ advert\langle\langle PE \times CHANNEL \times PE \rangle\rangle \\ & \mid\ rtr\langle\langle PE \times CHANNEL \times PE \rangle\rangle \\ & \mid\ msg\langle\langle PE \times CHANNEL \times PE \times MEM\_BLOCK \rangle\rangle \\ & \mid\ term\langle\langle PE \times THREAD \rangle\rangle \\ & \mid\ thrd\langle\langle PE \times THREAD \times (\mathbb{P}(CHANNEL \times PE)) \times \\ & \quad (\mathbb{P}(CHANNEL \times PE \times \mathbb{N}))\rangle\rangle \end{aligned}$$

focusm The focus message is the first of four kinds of message used during channel communication. When a thread requests to send and finds that the receiver thread is known to be on another processor it (the sender) sends a focus message to tell the receiver that it is waiting.

advert The advert message is used only during the first communication on a channel. When a thread requests to send and has no information about where the receiver is located it (the sender) sends an advertisement to each of the processors to tell them of its existence.

rtr The 'ready to receive' message is returned by receiver threads when they receive a focus or *advert* message.

msg The 'message' message contains the actual data to be transferred during a remote channel communication.

term The 'termination' message is not used in channel communication but in task synchronisation. If a parent thread creates children which subsequently migrate to a remote processor then when the children terminate a *term* message is sent over Centrenet to the parent's processor to inform the parent.

thrd The *thrd* message is used during thread migration. It contains various register values and control information.

The specification of the Testbed is primarily concerned with modelling the state and operations of TOS. In order to assist with the proofs, however, the specification defines a trace sequence which is built up as each state-changing operation is applied. The trace contains a concise summary of the operations applied (and their parameters) encoded as values of the type *OP*.

$$
\begin{array}{rl}
OP ::= & \mathsf{send}\langle\!\langle PE \times THREAD \times CHANNEL\rangle\!\rangle \\
| & \mathsf{receive}\langle\!\langle PE \times THREAD \times CHANNEL\rangle\!\rangle \\
| & \mathsf{advert}\langle\!\langle PE \times CHANNEL \times PE\rangle\!\rangle \\
| & \mathsf{focus}\langle\!\langle PE \times CHANNEL\rangle\!\rangle \\
| & \mathsf{rtr}\langle\!\langle PE \times CHANNEL\rangle\!\rangle \\
| & \mathsf{msg}\langle\!\langle PE \times CHANNEL\rangle\!\rangle \\
| & \mathsf{create}\langle\!\langle THREAD \times THREAD\rangle\!\rangle \\
| & \mathsf{sync}\langle\!\langle THREAD\rangle\!\rangle \\
| & \mathsf{terminate}\langle\!\langle THREAD\rangle\!\rangle \\
| & \mathsf{term\_msg}\langle\!\langle THREAD\rangle\!\rangle
\end{array}
$$

Some of the *OP* values have similar identifiers to those used in *TState* components (defined next) and *CNETMSG* types so the *OP* values are printed in a sans serif font so that they can be distinguished. The use of the *OP* types are not explained in detail but, for instance, the send *OP* is used to extend the trace during the application of the channel send operation.

4.1.2 State

I declare a schema called *TState* which represents part of the state that the Testbed may be in at any given moment. Formally, *parent*, *cntr* and *par_bd* are partial functions, e.g. *cntr* is a partial function from threads to counters, *ready* is a relation (between processors and threads) and *cnet* is a set (of Centrenet messages).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{TState}} \\
\hline
parent : THREAD \rightarrow\!\!\!\!\!\!+ \; THREAD \\
cntr : THREAD \rightarrow\!\!\!\!\!\!+ \; COUNTER \\
par\_bd : THREAD \rightarrow\!\!\!\!\!\!+ \; PE \\
ready : PE \leftrightarrow THREAD \\
cnet : \mathbb{P}\, CNETMSG \\
\hline
\end{array}
$$

Informally, *parent* gives for executing threads the parent thread that created them, *cntr* gives for each executing parent thread its counter, *par_bd* gives for every child the processor on which its parent is executing, *ready* associates with each processor the set of threads that are ready to execute on it and *cnet* models the processor interconnect (sending a message is modelled by adding the message to *cnet* and receiving a message is modelled by removing a message from *cnet*).

It is a convention in **Z** that each state schema should be followed by another schema which defines the initial value of the state schema, and this is what *TInit* does.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{TInit}} \\
\hline
TState \\
t? : THREAD \\
p? : PE \\
\hline
ready = \{p? \mapsto t?\} \\
par\_bd = \{t? \mapsto p?\} \\
parent = \{t? \mapsto null\_thread\} \\
cntr = \{null\_thread \mapsto (1, null\_thread)\} \\
cnet = \varnothing \\
\hline
\end{array}
$$

The reader may like to interpret this as follows. In the initial state execution of a user's program has just begun and there is just one thread $t?$ in the ready queue of one of the processors $p?$. The processor $p?$ is recorded as the location of $t?$'s parent. For convenience, the null thread is allocated the only counter and recorded as being the parent for $t?$. Centrenet has no messages to deliver.

4.1.3 State-changing operations

Now that the basic entities have been defined (threads, processors and counters) and I have declared the relationships (*parent*, *cntr*, *par_bd* and so on) between entities that are to be recorded in the state *TState*, the next part of the specification defines the operations which modify the state. The operations available to threads allow occam-like task control (as described in Section 2.1.3) to be implemented. Figure 4 shows an example where a parent thread creates three children and is suspended until they all terminate.

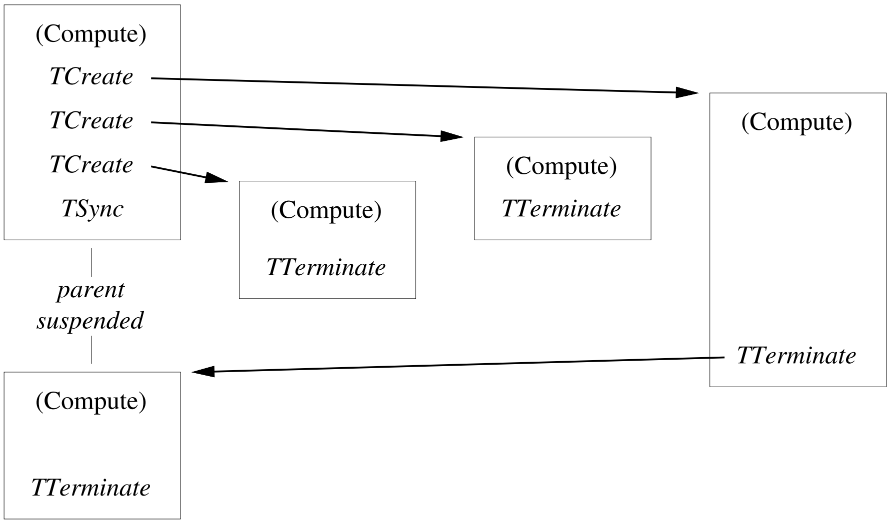

Figure 4: *An example of the occam-like task control on the Testbed.*

The first operation specified is called *TCreate* and models what happens when a parent thread calls the operating system and requests that a new child thread be created. The schema declaration $\Delta TState$ indicates that this schema changes that part of the Testbed state modelled by *TState*. The declarations $pt?, ct? : THREAD$ and $p? : PE$ are inputs to the operation and the reader should think of $pt?$ as the parent thread requesting the create service; $ct?$ as identifying the child thread to be created and $p?$ as being the processor on which the request is being made.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{TCreate}} \\
\hline
\Delta TState \\
pt?, ct? : THREAD \\
p? : PE \\
\hline
(\mathbf{let}\ p == \mathbf{if}\ pt? \in \operatorname{dom}(cntr)\ \mathbf{then}\ \mathit{first}(cntr\ pt?)\ \mathbf{else}\ 0 \bullet \\
\qquad cntr' = (\{ct?\} \mathbin{\lhd} cntr) \oplus \{pt? \mapsto (p+1, null\_thread)\}) \\
parent' = parent \cup \{ct? \mapsto pt?\} \\
ready' = ready \cup \{p? \mapsto ct?\} \\
par\_bd' = par\_bd \cup \{ct? \mapsto p?\} \\
\hline
\end{array}
$$

The first part of the schema predicate specifies how the counter part of *TState* is modified: if the parent does not have a counter ($pt? \notin \operatorname{dom}(cntr)$) then a new counter $(1, null\_thread)$ is associated with the parent thread; if the parent thread already has a counter ($pt? \in \mathrm{dom}(cntr)$) then the counter is incremented to $(\mathit{first}(cntr\ pt?) + 1, null\_thread)$. The $\{ct?\} \ndres cntr$ ensures that initially no counter is associated with the new child. The identity of the parent is recorded by $parent' = parent \cup \{ct? \mapsto pt?\}$. The new child is made ready to execute by adding it to the ready queue with $ready' = ready \cup \{p? \mapsto ct?\}$ and the parent's processor board is recorded for the child by $par\_bd' = par\_bd \cup \{ct? \mapsto p?\}$.

The next operation, *TSync*, models what happens once a parent has created some children and wants to be blocked until its children have finished executing. The inputs to the schema are $pt?$, the parent thread requesting to be blocked and $p?$, the processor on which the request is made. The schema predicates specify that *cntr* and *ready* are updated if the counter associated with the parent has a value greater than zero ($\mathit{first}(cntr\ pt?) > 0$); otherwise all the children must have terminated and no action is taken.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{TSync}} \\
\hline
\Delta TState \\
pt? : THREAD \\
p? : PE \\
\hline
\mathit{first}(cntr\ pt?) > 0 \Rightarrow \\
\qquad cntr' = cntr \oplus \{pt? \mapsto (\mathit{first}(cntr\ pt?), pt?)\} \wedge \\
\qquad ready' = ready \setminus \{p? \mapsto pt?\} \\
\hline
\end{array}
$$

If there are still children executing then the *cntr* function is updated by setting the parent's counter to the value $(\mathit{first}(cntr\ pt?), pt?)$ showing that the number of children is unchanged but that the parent thread $pt?$ is blocked on the counter. Simultaneously, the parent is removed from the ready queue by updating $ready'$ to $ready \setminus \{p? \mapsto pt?\}$.

The next schema, *TTerminate*, models a child thread requesting to terminate. The inputs are $ct?$ and $p?$ representing the child thread and the processor on which the child is executing. This time, there is also an output argument, $msgs!$ which is defined as the messages to be sent over Centrenet (or the empty set if none). This output argument allows the lower-level schema *TTerminate* to pass values back to the higher-level schema *FTerminate* (defined later on in Section 4.1.4).

The first two predicates delete any mappings involving the terminating thread from the *parent* and *par_bd* functions. The rest of the predicates are in two parts: the first part is applied when parent and child are executing on different processor boards ($par\_bd\ ct? \neq p?$) and the second part is applied when they are on the same processor board ($par\_bd\ ct? = p?$). If the threads are on different processor boards then the output argument is set to inform the parent on the remote processor of its child's termination. The child is removed from the ready queue for the processor $p?$ and any counters which might be owned by the child are deleted (so that they can be reused by other threads).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad TTerminate \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
\Delta TState \\
ct? : THREAD \\
p? : PE \\
msgs! : \mathbb{P}\, CNETMSG \\
\hline
parent' = \{ct?\} \lhd parent \\
par\_bd' = \{ct?\} \lhd par\_bd \\
(\mathbf{let}\ pt == parent\ ct? \bullet \\
par\_bd\ ct? \neq p? \Rightarrow \\
\quad msgs! = \{term(par\_bd\ ct?, pt)\} \land \\
\quad ready' = ready \setminus \{p? \mapsto ct?\} \land \\
\quad cntr' = \{ct?\} \lhd cntr \land \\
par\_bd\ ct? = p? \Rightarrow \\
\quad msgs! = \varnothing \land \\
\quad (\mathbf{let}\ c == cntr\ pt; \\
\quad r == ready \setminus \{p? \mapsto ct?\} \bullet \\
\quad first\ c = 1 \Rightarrow \\
\quad\quad (ready' = \mathbf{if}\ second\ c = pt\ \mathbf{then}\ r \oplus \{p? \mapsto pt\}\ \mathbf{else}\ r \land \\
\quad\quad cntr' = (\{ct?\} \lhd cntr) \oplus \{pt \mapsto (0, null\_thread)\}) \land \\
\quad first\ c > 1 \Rightarrow \\
\quad\quad (ready' = r \land \\
\quad\quad cntr' = (\{ct?\} \lhd cntr) \oplus \{pt \mapsto (first\ c - 1, second\ c)\}))) \\
\hline
\end{array}
$$

If the threads are on the same processor board then the output argument $msgs!$ is set to $\varnothing$ to indicate that no messages need to be transmitted to remote parents. For convenience the 'temporary variable' c is created and set to the parent's counter and the variable r is created and set to the contents of the ready queue with the child removed. There are then two possible cases: $first\ c = 1$ meaning that the last child is terminating and $first\ c > 1$ meaning that there are still children executing. If the last child is terminating then it is removed from the ready queue, any counters owned by it are deleted and the parent's counter is set to zero. If there was a parent waiting for the counter then the parent is returned to the ready queue. If the terminating child is not the last child then it is removed from the ready queue and the parent's counter is decremented.

Finally, here is the $TTerm\_msg$ schema which is responsible for receiving a 'termination message'. Termination messages are sent from terminating children to parents when the parents are executing on different processors. The input arguments are $pt?$, which indicates the parent, and $p?$ which gives the location of the parent. In the predicate part of $TTerm\_msg$ a temporary variable is created and set to the parent's counter. The value of this counter is tested and, as in the $TTerminate$ schema, if the counter indicates that the last child is terminating then the parent is returned to the ready queue and its counter reset; otherwise the parent's counter is decremented. Despite the convention that state components not mentioned in an operation schema are assumed to retain their original value, the predicate $ready' = ready$ is stated here for reasons of clarity.

$$
\begin{array}{l}
\textit{TTerm\_msg} \\
\hline
\Delta \textit{TState} \\
pt? : \textit{THREAD} \\
p? : \textit{PE} \\
\hline
\textbf{let } c == \textit{cntr}\ pt? \bullet \\
\textit{first}\ c = 1 \Rightarrow \\
\qquad (\textit{ready}' = \textbf{if } \textit{second}\ c = pt? \textbf{ then } \textit{ready} \oplus \{p? \mapsto pt?\} \textbf{ else } \textit{ready} \wedge \\
\qquad \textit{cntr}' = \textit{cntr} \oplus \{pt? \mapsto (0, \textit{null\_thread})\}) \wedge \\
\textit{first}\ c > 1 \Rightarrow (\textit{ready}' = \textit{ready} \wedge \\
\qquad \textit{cntr}' = \textit{cntr} \oplus \{pt? \mapsto (\textit{first}\ c - 1, \textit{second}\ c)\})
\end{array}
$$

4.1.4 Model of execution

So far in this section I have declared some basic entities for the model, defined a representation for the state and presented four operations which update the state. I now present four, higher-level 'framing schemas' which build on top of the preceding definitions and give an explanation of *when*, rather than *what*, operations on *TState* are performed.

The life-cycle of Testbed threads is depicted in Figure 5. All threads begin in the unborn state and remain there until the create operation moves them into the ready state. In accordance with the scheduling procedure, threads are taken from the ready state one-by-one and allowed to execute on the CPU until: an error or termination request occurs; or the time-slice ends; or a request for an operating system service occurs. Terminating or erroneous threads make no further state changes, threads at the end of their time-slice return immediately to the ready state and threads requesting system services return to the ready state once the service has completed.

For simplicity, not all states and transitions are modelled. For the purposes of the occam task model I am concerned only with the ready and waiting-to-synchronise states and with the create, request-service, service-completed and terminate transitions.

The specification models the programs executed by threads as sequences of instructions where each instruction has a fixed type and, sometimes, parameters. The **Z** construct for free types is used to define *INSTRUCTION* as either: a create instruction parameterised on the child to be created; a synchronise instruction; a terminate instruction; or a send or receive parameterised by channel and memory block for the message. (The send and receive instructions will be used later on in Section 4.2.3 when I specify the schemas for channel communication operations.)

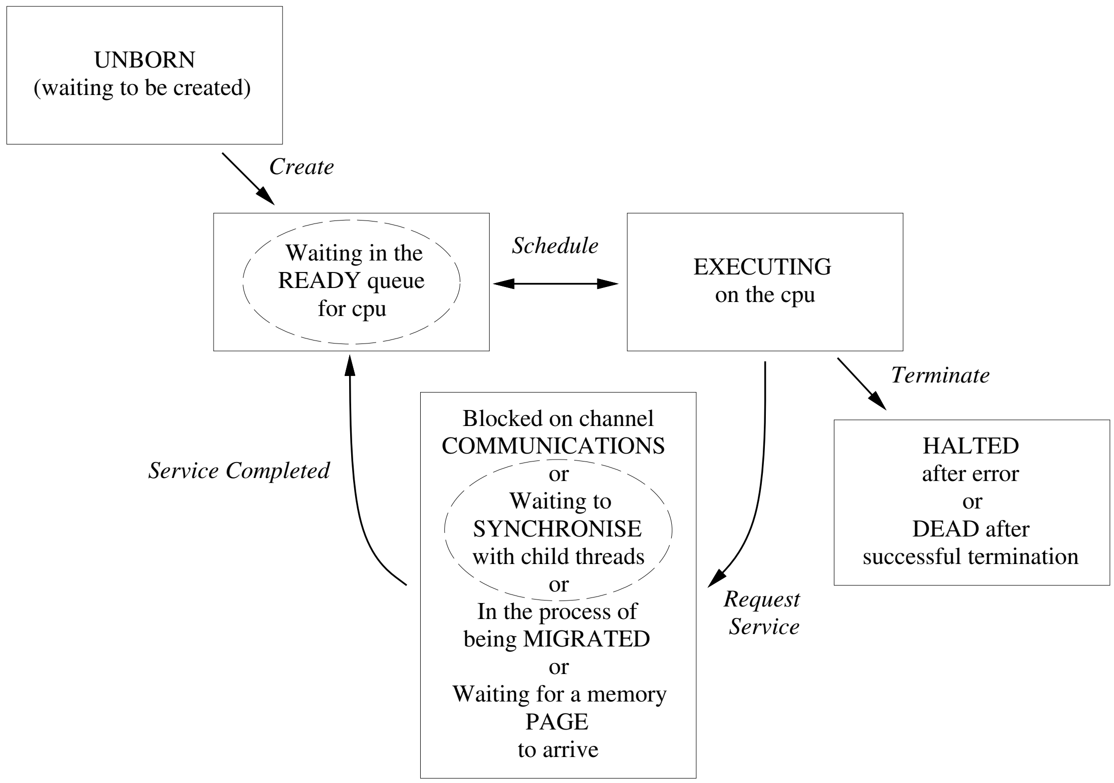

Figure 5: *The life-cycle of threads on the Testbed.*

$$
\begin{aligned}
INSTRUCTION ::= \; & create\langle\!\langle THREAD \rangle\!\rangle \\
| \; & sync \\
| \; & terminate \\
| \; & send\langle\!\langle CHANNEL \times MEM\_BLOCK \rangle\!\rangle \\
| \; & receive\langle\!\langle CHANNEL \times MEM\_BLOCK \rangle\!\rangle
\end{aligned}
$$

I model more of the state of the Testbed by defining the schema *FState* as everything in *TState* plus a set of program counters (one per thread), programs (one per thread) and operation traces (one trace for all threads).

$$
\begin{array}{|l}
\hline
\textit{FState} \\
\textit{TState} \\
pc : THREAD \rightarrowtail\!\!\!\!\!\!\leftarrow \mathbb{N} \\
program : THREAD \rightarrowtail\!\!\!\!\!\!\leftarrow \operatorname{seq} INSTRUCTION \\
trace : \operatorname{seq} OP \\
\hline
\end{array}
$$

The operation schemas defined in this section will modify *FState* by executing instructions from these programs. The function *pc* gives for each executing thread the number of the next instruction to be executed in its program and the *trace* is the sequence of all operations performed so far by the Testbed processors.

The *FInit* schema defines the initial value of *FState* as all the predicates in *TInit* plus the condition that the initial thread's program counter is set to 1 plus the condition that the null thread does not have a program. It is assumed that the programs associated with the other threads include sensible combinations of instructions—the proofs in Section 6 state exactly what is meant by a 'well-behaved program'. The creation of $t?$ is the only thing recorded in the trace. The application of $TInit[t/t?, p/p?]$ illustrates the use of substitution which is typical in framing schemas: the new variable t declared in *FInit* is to be substituted for the old $t?$ declared in *TInit* and the new p substituted for $p?$.

FInit

$FState$

$$
\begin{array}{l}
\exists_1\, t : THREAD \setminus \{null\_thread\};\ p : PE \bullet \\
TInit[t/t?, p/p?] \land \\
pc = \{t \mapsto 1\} \land trace = \langle \mathsf{create}(null\_thread, t)\rangle \\
null\_thread \notin \mathrm{dom}(program)
\end{array}
$$

The framing schema, *FCreate*, for the create operation states that if a parent pt is in the ready list of processor p and the current instruction in its program is to create a child thread ct, then the *TCreate* operation should be applied. Assuming that the preconditions of *TCreate* are satisfied and that *TCreate* updates *TState*, then the *FState* is modified by incrementing the program counter for the parent, initialising the program counter for the child and augmenting the trace.

FCreate

$\Delta FState$

$$
\begin{array}{l}
\exists\, pt, ct : THREAD;\ p : PE \bullet \\
(p, pt) \in ready \land \\
create(ct) = program\ pt(pc\ pt) \land \\
TCreate[pt/pt?, ct/ct?, p/p?] \land \\
pc' = pc \oplus \{pt \mapsto pc\ pt + 1, ct \mapsto 1\} \land \\
trace' = trace \frown \langle \mathsf{create}(pt, ct)\rangle
\end{array}
$$

The *FSync* schema follows a similar format to *FCreate*. A parent thread pt in the ready state executes the instruction to synchronise with its children thus causing the *TSync* schema to be applied, its program counter to be incremented and the trace to be augmented to record the synchronisation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FSync}} \\
\Delta \mathit{FState} \\
\hline
\exists\, pt : \mathit{THREAD};\ p : \mathit{PE} \bullet \\
(p, pt) \in \mathit{ready} \land \\
\mathit{sync} = \mathit{program}\, pt(\mathit{pc}\, pt) \land \\
\mathit{TSync}[pt/pt?, p/p?] \land \\
pc' = pc \oplus \{pt \mapsto pc\, pt + 1\} \land \\
\mathit{trace}' = \mathit{trace} \frown \langle \mathsf{sync}(pt) \rangle \\
\hline
\end{array}
$$

In *FTerminate* the *TTerminate* schema defines the output argument *msgs*!. If the parent of the terminating child is on the same processor then *msgs*! is the empty set and the trace is augmented, otherwise the parent is on a different processor, a *term* is transmitted and the trace is not changed.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FTerminate}} \\
\Delta \mathit{FState} \\
\hline
\exists\, ct : \mathit{THREAD};\ p : \mathit{PE};\ \mathit{msgs} : \mathbb{P}\, \mathit{CNETMSG} \bullet \\
(p, ct) \in \mathit{ready} \land \\
\mathit{terminate} = \mathit{program}\, ct(\mathit{pc}\, ct) \land \\
\mathit{TTerminate}[ct/ct?, p/p?, \mathit{msgs}/\mathit{msgs}!] \land \\
\mathit{cnet}' = \mathit{cnet} \cup \mathit{msgs} \land \\
pc' = pc \oplus \{ct \mapsto pc\, ct + 1\} \land \\
\mathit{trace}' = \mathbf{if}\ \mathit{msgs} = \varnothing\ \mathbf{then}\ \mathit{trace} \frown \langle \mathsf{terminate}(ct) \rangle\ \mathbf{else}\ \mathit{trace} \\
\hline
\end{array}
$$

The final schema *FTerm_msg* shows what happens at the remote site when the *FTerminate* schema sends a *term* message over Centrenet: $\exists\, pp : PE;\ pt : THREAD \bullet term(pp, pt) \in cnet$ represents the situation where Centrenet delivers a terminate message for parent thread *pt* to processor *pp*. The effect of the schema is to apply the *TTerm_msg* schema and delete the *term* message from *cnet* to indicate that it has been obeyed. The trace is updated to record the termination.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FTerm\_msg}} \\
\Delta \mathit{FState} \\
\hline
\exists\, pp : \mathit{PE};\ pt : \mathit{THREAD} \bullet \\
\mathit{term}(pp, pt) \in \mathit{cnet} \land \\
\mathit{TTerm\_msg}[pp/p?, pt/pt?] \land \\
\mathit{cnet}' = \mathit{cnet} \setminus \{\mathit{term}(pp, pt)\} \land \\
\mathit{trace}' = \mathit{trace} \frown \langle \mathsf{term\_msg}(pt) \rangle \\
\hline
\end{array}
$$

4.2 Channel Design

This section introduces two new objects (channels and memory blocks) which it uses to define that part of the Testbed state concerned with communication. Six channel operations are presented and six higher-level framing schemas.

4.2.1 Basic objects and state

I declare two more given sets, $CHANNEL$ the set of all Testbed communication channels, and MEM_BLOCK the set of memory blocks. The memory blocks are used when modelling communications to give the idea of message transfer being effected by a memory-to-memory block copy.

$[CHANNEL, MEM\_BLOCK]$

Here is the schema $CState$ which models the part of the Testbed state to do with channel communications. Actually, $CState$ includes everything in $TState$ as well, so it also models thread state.

$$\begin{array}{|l}
\multicolumn{1}{l}{CState} \\
\hline
TState \\
scontrol, rcontrol, sender, receiver : PE \rightarrowtail\!\!\!\!\!\!+ (CHANNEL \rightarrowtail\!\!\!\!\!\!+ THREAD) \\
sloc, rloc : PE \rightarrowtail\!\!\!\!\!\!+ (CHANNEL \rightarrowtail\!\!\!\!\!\!+ PE) \\
focus, used : PE \leftrightarrow CHANNEL \\
buffer : PE \rightarrowtail\!\!\!\!\!\!+ (CHANNEL \rightarrowtail\!\!\!\!\!\!+ MEM_BLOCK) \\
\hline
\end{array}$$

The functions used to model the channel state have quite complicated semantics, but at a reasonably abstract level they can be described as follows:

scontrol: The maplet $p \mapsto \{c \mapsto t\}$ exists in *scontrol* if and only if the processor p has a thread t blocked, waiting to send on c. The *scontrol* function has two uses: it provides a place to store the thread while not in the ready queue and it can be checked to see if the channel is in use.

rcontrol: This function is used in the same way as *scontrol* but for storing blocked receiver threads.

sender: The maplet $p \mapsto \{c \mapsto t\}$ is present here if and only if the thread t has, since the beginning of the program execution, requested to send on channel c while executing on p (or if it has communicated on c and then been migrated to p). The *sender* function is used during thread migration to check which channels a thread has sent on.

receiver: This function is similar to *sender*, except that it records the channels a thread has received on.

sloc: The maplet $p_1 \mapsto \{c \mapsto p_2\}$ is defined in this function if and only if the processor p_1 is storing p_2 as the last known location for the sender for channel c. If undefined, then p_1 knows nothing about the sender's location.

rloc: This function is used in the same way as *sloc*, except that where defined it indicates the location of the receiver.

focus: Synchronisation during communication is enforced by having the sender produce a 'focus' (essentially, just a token) when it requests to send and requiring the receiver to consume that same focus before completing the communication. The focus for a channel c is able to travel between processors (so that senders can synchronise with remote receivers) but when and only when it stops at a particular processor p is the function *focus* defined for the maplet $p \mapsto c$.

used: The function defines the maplet $p \mapsto c$ if and only if a sender or receiver executing on processor p has, since the beginning of the program execution, requested to communicate on channel c. The function is needed to distinguish the first communication over a channel when the sender and receiver still need to locate each other, from subsequent communications when they know where each other are.

buffer: This is a place for blocked threads to store their message or empty buffer while they wait for communication to complete.

The initial channel state *CInit* is defined as the initial thread state extended to model the situation where no channel communication has occurred yet—hence all functions are undefined.

$$
\begin{array}{|l}
\hline
\textit{CInit} \\
\hline
\textit{CState} \\
\hline
\exists\, t : \textit{THREAD};\ p : \textit{PE} \bullet \textit{TInit}[t/t?, p/p?] \\
\textit{scontrol} = \textit{rcontrol} = \textit{sender} = \textit{receiver} = \varnothing \\
\textit{sloc} = \textit{rloc} = \varnothing \wedge \textit{focus} = \textit{used} = \varnothing \wedge \textit{buffer} = \varnothing \\
\hline
\end{array}
$$

4.2.2 State-changing operations

There are six channel operation schemas. The first two are applied when threads request to send or receive on a channel, the other four are applied on reception of various kinds of Centrenet message involved in remote channel communication.

The *CSend* schema models the situation where a sender thread *st*? executing on 'sender processor' *sp*? requests to send the message in its message buffer *m*?

over channel $c?$. The schema also has an output argument $msgs!$ which is set to a (possibly empty) set of messages to be transmitted over Centrenet.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CSend \quad}} \\
\Delta CState \\
st? : THREAD \\
c? : CHANNEL \\
sp? : PE \\
m? : MEM\_BLOCK \\
msgs! : \mathbb{P}\, CNETMSG \\
\hline
sloc' = sloc \oplus \{sp? \mapsto \{c? \mapsto sp?\}\} \\
sender' = sender \oplus \{sp? \mapsto \{c? \mapsto st?\}\} \\
c? \in \mathrm{dom}(rcontrol\ sp?) \Rightarrow \\
\qquad (\mathbf{let}\ rt == rcontrol\ sp?\ c? \bullet \\
\qquad (\mathbf{let}\ b == buffer\ sp?\ c? \bullet MEM\_COPY[m?/from?, b/to?]) \wedge \\
\qquad ready' = ready \cup \{sp? \mapsto rt\} \wedge \\
\qquad rcontrol' = rcontrol \setminus \{sp? \mapsto \{c? \mapsto rt\}\} \wedge \\
\qquad used' = used \cup \{sp? \mapsto c?\} \wedge \\
\qquad msgs! = \varnothing) \\
c? \notin \mathrm{dom}(rcontrol\ sp?) \Rightarrow \\
\qquad (buffer' = buffer \oplus \{sp? \mapsto \{c? \mapsto m?\}\} \wedge \\
\qquad ready' = ready \setminus \{sp? \mapsto st?\} \wedge \\
\qquad scontrol' = scontrol \oplus \{sp? \mapsto \{c? \mapsto st?\}\} \wedge \\
\qquad c? \in \mathrm{dom}(rloc\ sp?) \Rightarrow \\
\qquad\qquad (focus', msgs!) = \mathbf{if}\ rloc\ sp?\ c? = sp?\ \mathbf{then}\ (focus \cup \{sp? \mapsto c?\}, \varnothing) \\
\qquad\qquad\qquad \mathbf{else}\ (focus, \{focusm(rloc\ sp?\ c?, c?, sp?)\})) \wedge \\
\qquad (c? \notin \mathrm{dom}(rloc\ sp?) \Rightarrow \\
\qquad\qquad (focus' = focus \cup \{sp? \mapsto c?\} \wedge \\
\qquad\qquad msgs! = \{p : PE \setminus \{sp?\} \bullet advert(p, c?, sp?)\})) \\
\hline
\end{array}
$$

The location of the sender is stored in *sloc* for the use of the receiver (if it is local). The identity of the sender is stored in *sender* so that during migration the Testbed can tell which channels a given thread has sent on. The rest of the predicates are in two parts depending on whether there is a local, receiver thread already waiting to receive on the channel ($c? \in \mathrm{dom}(rcontrol\ sp?)$) or not ($c? \notin \mathrm{dom}(rcontrol\ sp?)$).

If there is a local, waiting receiver then a temporary variable *rt* is defined to be the receiver thread and a variable b to be the receiver's buffer. The detail of the transfer of the message is hidden inside the *MEM_COPY* schema (not defined in the specification). The receiver thread is unblocked ($rcontrol' = rcontrol \setminus \{sp? \mapsto \{c? \mapsto rt\}\}$) and returned to the ready queue ($ready' = ready \cup \{sp? \mapsto rt\}$). The channel is marked as having been used. It is to be understood that a channel focus has been produced and immediately consumed.

If there is no local, waiting receiver then the sender's buffer is recorded in *buffer* for ease of retrieval later and the sender is deleted from the ready queue and blocked on the channel. There are now two, alternative cases depending on whether there is a last known location for the receiver. If $c? \in \mathrm{dom}(rloc\ sp?)$ then the receiver was last at $rloc\ sp?\ c?$ so the focus is sent there by putting $focusm(rloc\ sp?\ c?, c?, sp?)$ in $msgs!$ or the state component *focus* is updated locally if the receiver was last known on $sp?$.

If nothing is known about the location of the receiver ($c? \notin \mathrm{dom}(rloc\ sp?)$) then the focus is stored locally by $focus' = focus \cup \{sp? \mapsto c?\}$ and a series of messages are sent advertising the presence of the waiting sender. These messages are sent to all processors except the one where the sender is waiting ($p : PE \setminus \{sp?\}$).

$$
\begin{array}{|l}
\hline
\textit{CReceive} \\
\Delta CState \\
rt? : THREAD \\
c? : CHANNEL \\
rp? : PE \\
b? : MEM\_BLOCK \\
msgs! : \mathbb{P}\, CNETMSG \\
\hline
rloc' = rloc \oplus \{rp? \mapsto \{c? \mapsto rp?\}\} \\
receiver' = receiver \oplus \{rp? \mapsto \{c? \mapsto rt?\}\} \\
c? \in \mathrm{dom}(scontrol\ rp?) \Rightarrow \\
\quad (\mathbf{let}\ st == scontrol\ rp?\ c? \bullet \\
\quad (\mathbf{let}\ m == buffer\ rp?\ c? \bullet MEM\_COPY[m/from?, b?/to?]) \wedge \\
\quad ready' = ready \cup \{rp? \mapsto st\} \wedge \\
\quad scontrol' = scontrol \setminus \{rp? \mapsto \{c? \mapsto st\}\} \wedge \\
\quad used' = used \cup \{rp? \mapsto c?\} \wedge \\
\quad focus' = focus \setminus \{rp? \mapsto c?\} \wedge \\
\quad msgs! = \varnothing) \\
c? \notin \mathrm{dom}(scontrol\ rp?) \Rightarrow \\
\quad (buffer' = buffer \oplus \{rp? \mapsto \{c? \mapsto b?\}\} \wedge \\
\quad ready' = ready \setminus \{rp? \mapsto rt?\} \wedge \\
\quad rcontrol' = rcontrol \oplus \{rp? \mapsto \{c? \mapsto rt?\}\} \wedge \\
\quad (focus', msgs!) = \mathbf{if}\ rp? \mapsto c? \in focus \\
\qquad \mathbf{then}\ (focus \setminus \{rp? \mapsto c?\}, \{rtr(sloc\ rp?\ c?, c?, rp?)\}) \\
\qquad \mathbf{else}\ (focus, \\
\qquad\quad \mathbf{if}\ c? \in \mathrm{dom}(sloc\ rp?) \wedge sloc\ rp?\ c? \neq rp? \wedge rp? \mapsto c? \notin used \\
\qquad\quad \mathbf{then}\ \{rtr(sloc\ rp?\ c?, c?, rp?)\}\ \mathbf{else}\ \varnothing)) \\
\hline
\end{array}
$$

The *CReceive* schema models the situation where a receiver thread $rt?$ executing on 'receiver processor' $rp?$ requests to receive a message into its buffer $b?$. The send and receive schemas are similar in the case where a thread requesting communication finds the other thread already waiting. In the case that the sender finds no receiver waiting, its actions are based on whether it knows the receiver's location and, if so, whether the receiver is local or remote. In the case that the receiver finds no waiting sender, it bases its actions on whether there is an indication of a waiting sender's focus or advertisement, or no such indication.

The location of the receiver is stored in *rloc* and the identity of the receiver in *receiver*. The predicates are then in two parts, depending on whether there is a local, waiting sender. If there is a waiting sender then a temporary variable *st* is defined to be the sender and a variable m to be the sender's message. The message transfer is effected by MEM_COPY, the sender unblocked from the channel and returned to the ready queue. The used flag is set for the channel and the *focus* flag is forced to undefined regardless of its previous state. Again, it is to be understood that a focus was produced by the sender and immediately consumed by the receiver.

If there is no local, waiting sender then the receiver's buffer is recorded, the receiver is deleted from the ready queue and blocked on the channel. There are now two cases depending on whether a focus for the channel is present. If a focus is present then an *rtr* message is sent to the sender at the location specified in $sloc\ rp?\ c?$ (I will prove later that this is indeed an appropriate destination for the message). The focus is consumed by the receiver.

If no focus is found then a decision is made whether to send an *rtr* or not. Initially, all channels have undefined sender location values and undefined *used* flags. A channel that has received an advertisement from a remote sender but has not been used, has a defined *sloc* but an undefined *used* flag—a receiver should send an *rtr* in this case. A channel that has not received an advertisement, has been used already, or which has a local sender, should not send an *rtr*.

The *CAdvert* schema models the situation on the Testbed where an advertisement for a channel sent by a sender on processor $sp?$ arrives at a processor $rp?$. The predicates simply require that the sender location should be defined (or updated) and that an *rtr* message be returned if there is a waiting receiver.

CAdvert

$\Delta CState$
$rp?, sp? : PE$
$c? : CHANNEL$
$msgs! : \mathbb{P}\, CNETMSG$

$sloc' = sloc \oplus \{rp? \mapsto \{c? \mapsto sp?\}\}$
$msgs! = \textbf{if } c? \in \mathrm{dom}(rcontrol\ rp?)\ \textbf{then } \{rtr(sp?, c?, rp?)\}\ \textbf{else } \varnothing$

The *CFocus* schema models the situation where a focus for a channel, originally from the $sp?$ processor, arrives at a new processor $rp?$.

$$
\begin{array}{|l}
\hline
\textit{CFocus} \\
\Delta \textit{CState} \\
rp?, sp? : PE \\
c? : \textit{CHANNEL} \\
msgs! : \mathbb{P}\, \textit{CNETMSG} \\
\hline
sloc' = sloc \oplus \{rp? \mapsto \{c? \mapsto sp?\}\} \\
c? \in \operatorname{dom}(rloc\ rp?) \land rloc\ rp?\ c? \neq rp? \Rightarrow \\
\qquad msgs! = \{focusm(rloc\ rp?\ c?, c?, sp?)\} \\
(c? \notin \operatorname{dom}(rloc\ rp?) \lor rloc\ rp?\ c? = rp?) \land c? \notin \operatorname{dom}(rcontrol\ rp?) \Rightarrow \\
\qquad (focus' = focus \cup \{rp? \mapsto c?\} \land \\
\qquad msgs! = \varnothing) \\
(c? \notin \operatorname{dom}(rloc\ rp?) \lor rloc\ rp?\ c? = rp?) \land c? \in \operatorname{dom}(rcontrol\ rp?) \Rightarrow \\
\qquad msgs! = \{rtr(sp?, c?, rp?)\} \\
\hline
\end{array}
$$

The predicates specify that the sender location is updated and that the focus is either forwarded, stored, or consumed. Forwarding of the focus occurs if the channel's *rloc* points to another site—this would occur if the receiver had migrated away from $rp?$. The focus is stored if *rloc* is undefined (or points to $rp?$) and there is no waiting receiver. Otherwise, if *rloc* does not point to another site and there is a waiting sender then an *rtr* is returned to the sender and the focus is consumed.

The *CRtr* schema models the situation where an *rtr* message for channel $c?$ arrives at a sender's processor $sp?$ from the receiver's processor $rp?$. The receiver's location is updated by overwriting *rloc* with $\{sp? \mapsto \{c? \mapsto rp?\}\}$ and the output variable $msgs!$ is set to contain the message data which was stored in *buffer* during the preceding application of *CSend*. The output argument $st!$ is defined and the sender thread is unblocked by moving it from *scontrol* to *ready*. Regardless of their previous values, the flag *focus* is forced to be undefined for channel $c?$ at processor $sp?$ and the flag *used* is forced to be defined for $c?$ at $sp?$.

$$
\begin{array}{|l}
\hline
\textit{CRtr} \\
\Delta \textit{CState} \\
sp?, rp? : PE \\
c? : \textit{CHANNEL} \\
st! : \textit{THREAD} \\
msgs! : \mathbb{P}\, \textit{CNETMSG} \\
\hline
rloc' = rloc \oplus \{sp? \mapsto \{c? \mapsto rp?\}\} \\
msgs! = \{msg(rp?, c?, sp?, buffer\ sp?\ c?)\} \\
st! = scontrol\ sp?\ c? \\
ready' = ready \cup \{sp? \mapsto st!\} \\
scontrol' = scontrol \setminus \{sp? \mapsto \{c? \mapsto st!\}\} \\
focus' = focus \setminus \{sp? \mapsto c?\} \\
used' = used \cup \{sp? \mapsto c?\} \\
\hline
\end{array}
$$

The *CMsg* schema models the situation where a receiver on processor $rp?$ receives a data message from $sp?$. The copying of the message into the receiver's memory area is suggested by MEM_COPY and the receiver, defined by the output argument $rt!$, is unblocked and returned to the ready queue. The location of the sender is updated and the *used* flag set for the channel $c?$ on the receiver's processor.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{CMsg}\ }} \\
\Delta \mathit{CState} \\
rp?, sp? : \mathit{PE} \\
c? : \mathit{CHANNEL} \\
m? : \mathit{MEM\_BLOCK} \\
rt! : \mathit{THREAD} \\
\hline
(\mathbf{let}\ b == \mathit{buffer}\ rp?\ c? \bullet \mathit{MEM\_COPY}[m?/\mathit{from}?, b/\mathit{to}?]) \\
\mathit{sloc}' = \mathit{sloc} \oplus \{rp? \mapsto \{c? \mapsto sp?\}\} \\
rt! = \mathit{rcontrol}\ rp?\ c? \\
\mathit{ready}' = \mathit{ready} \cup \{rp? \mapsto rt!\} \\
\mathit{rcontrol}' = \mathit{rcontrol} \setminus \{rp? \mapsto \{c? \mapsto rt!\}\} \\
\mathit{used}' = \mathit{used} \cup \{rp? \mapsto c?\} \\
\hline
\end{array}
$$

4.2.3 Model of execution

I have now declared the basic channel communication entities, defined a representation for the channel state and presented six operations which update this state. In this section I present framing schemas for the operations to specify *when* (rather than *how*) the channel state is changed.

Figures 6 and 7 summarise the sequences of channel operations allowed. The transitions are labelled with the names of framing schemas presented later in this section and indexed by the number of the processor at which they are applied (i and j are two arbitrarily selected processors such that $i \neq j$). The two vertices in bold indicate states where there is no ongoing communication. The eight cases may be interpreted as follows:

Case 1: Two threads executing on the same processor (number i) share a local channel. The sender communicates first (the *FSend.i* transition).

Case 2: This case is the same as Case 1, except that it is the receiver which communicates first (the *FReceive.i* transition).

Case 3: Two threads share a channel but the sender is executing on processor i and the receiver on processor j. The sender communicates first.

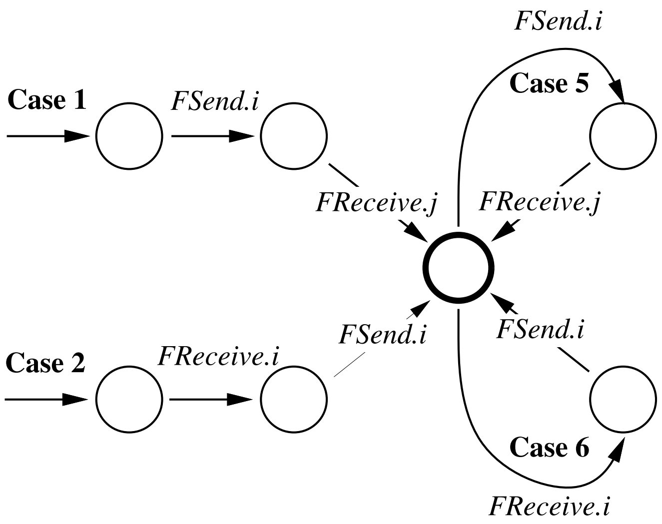

Figure 6: *Allowed sequences of local channel operations.*

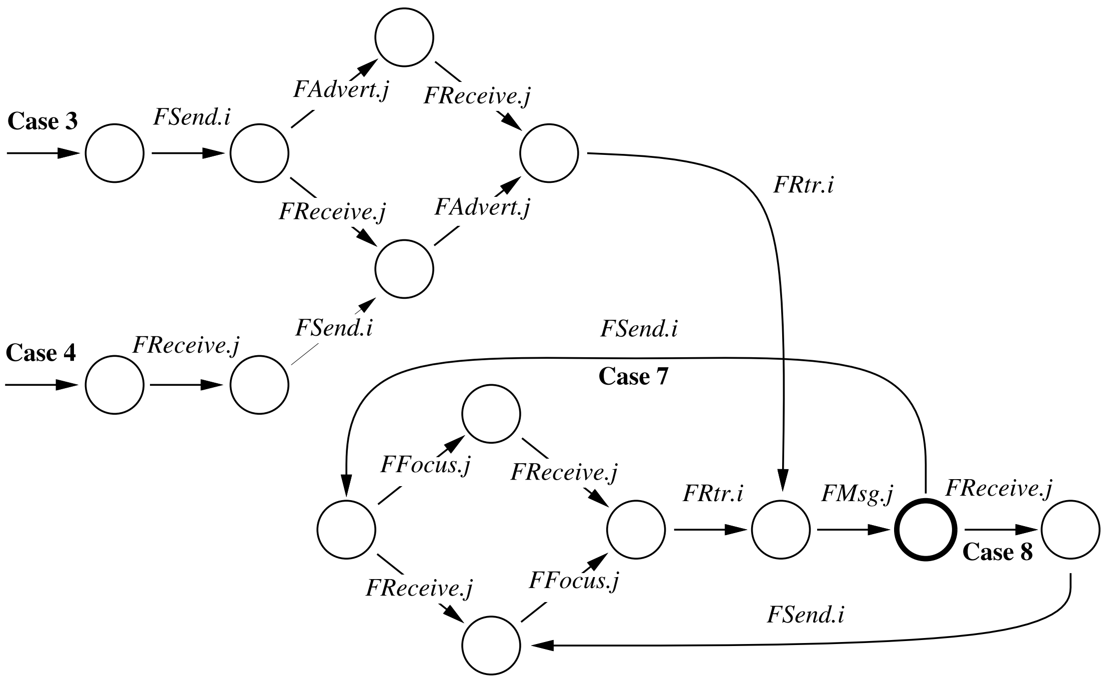

Figure 7: *Allowed sequences of remote channel operations.*

Case 4: This case is the same as Case 3 except that it is the receiver which communicates first.

Case 5: Two threads communicate for a second or subsequent time over a local channel. The sender communicates first.

Case 6: As Case 5 except that the receiver communicates first.

Case 7: Two threads on different processors communicate for the second or subsequent time. The sender communicates first.

Case 8: As Case 7 except that the receiver communicates first.

The *FState* and *FInit* schemas given next extend and replace *FState* and *FInit* previously defined in Section 4.1.4, on pages 23 and 24. The new *FState* schema includes the channel state *CState* and the new *FInit* schema specifies the initial channel state *CInit*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FState}} \\
\hline
\textit{TState} \\
\textit{program} : \textit{THREAD} \mathrel{+\!\!\!\!\twoheadrightarrow} \operatorname{seq} \textit{INSTRUCTION} \\
\textit{CState} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FInit}} \\
\hline
\textit{FState} \\
\hline
\textit{TInit} \\
\textit{null\_thread} \notin \operatorname{dom}(\textit{program}) \\
\textit{CInit} \\
\hline
\end{array}
$$

The framing schema for the send operation, *FSend*, states that if sender *st* is in the ready queue of processor *sp* and the first instruction in its program is to send a message *m* over channel *c*, then the *CSend* operation should be applied. Assuming that *CSend* updates *CState*, *FState* is modified by adding any messages in *msgs* to *cnet*, by incrementing the sender's program counter if the communication completes and by adding the tuple (send, st, sp) to the end of the trace.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FSend}} \\
\hline
\Delta \textit{FState} \\
\hline
\exists\, c : \textit{CHANNEL};\ m : \textit{MEM\_BLOCK};\ st : \textit{THREAD};\ sp : \textit{PE}; \\
\textit{msgs} : \mathbb{P}\, \textit{CNETMSG} \bullet \\
(sp, st) \in \textit{ready} \wedge \textit{send}(c, m) = \textit{program}\ st(pc\ st) \wedge \\
\textit{CSend}[st/st?, c/c?, sp/sp?, m/m?, \textit{msgs}/\textit{msgs}!] \wedge \\
\textit{cnet}' = \textit{cnet} \cup \textit{msgs} \wedge \\
pc' = \mathbf{if}\ (sp, st) \in \textit{ready}'\ \mathbf{then}\ pc \oplus \{st \mapsto pc\ st + 1\}\ \mathbf{else}\ pc \wedge \\
\textit{trace}' = \textit{trace} \frown \langle \mathsf{send}(sp, st, c) \rangle \\
\hline
\end{array}
$$

The framing schema *FReceive* states that if receiver *rt* is in the ready queue of processor *rp* and the first instruction in the program is to receive a message into buffer *b* over channel *c*, then the *CReceive* operation should be applied, new messages submitted to Centrenet, the program counter incremented if the communication completes and the trace extended.

$$
\begin{array}{|l}
\textit{FReceive} \\
\hline
\Delta FState \\
\hline
\exists\, c : CHANNEL;\ b : MEM\_BLOCK;\ rt : THREAD;\ rp : PE; \\
msgs : \mathbb{P}\, CNETMSG \bullet \\
(rp, rt) \in ready \land receive(c, b) = program\ rt(pc\ rt) \land \\
CReceive[rt/rt?, c/c?, rp/rp?, b/b?, msgs/msgs!] \land \\
pc' = \mathbf{if}\ (rp, rt) \in ready'\ \mathbf{then}\ pc \oplus \{rt \mapsto pc\ rt + 1\}\ \mathbf{else}\ pc \land \\
cnet' = cnet \cup msgs \land trace' = trace \frown \langle \mathsf{receive}(rp, rt, c)\rangle \\
\hline
\end{array}
$$

The rest of framing schemas, *FAdvert*, *FFocus*, *FRtr* and *FMsg* are similar in format to *FReceive* except that they require a particular message to be delivered by Centrenet rather than the next program instruction to be of a particular kind.

$$
\begin{array}{|l}
\textit{FAdvert} \\
\hline
\Delta FState \\
\hline
\exists\, c : CHANNEL;\ sp, rp : PE;\ st : THREAD;\ msgs : \mathbb{P}\, CNETMSG \bullet \\
advert(rp, c, sp) \in cnet \land \\
CAdvert[rp/rp?, c/c?, sp/sp?, msgs/msgs!] \land \\
cnet' = (cnet \setminus \{advert(rp, c, sp)\}) \cup msgs \land \\
trace' = trace \frown \langle \mathsf{advert}(rp, c, sp)\rangle \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{FFocus} \\
\hline
\Delta FState \\
\hline
\exists\, c : CHANNEL;\ sp, rp : PE;\ st : THREAD;\ msgs : \mathbb{P}\, CNETMSG \bullet \\
focusm(rp, c, sp) \in cnet \land \\
CFocus[rp/rp?, c/c?, sp/sp?, msgs/msgs!] \land \\
trace' = trace \frown \langle \mathsf{focus}(rp, c)\rangle \land \\
cnet' = (cnet \setminus \{focusm(rp, c, sp)\}) \cup msgs \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FRtr}} \\ \hline
\Delta FState \\ \hline
\exists\, c : CHANNEL;\ sp, rp : PE;\ msgs : \mathbb{P}\, CNETMSG;\ rt, st : THREAD \bullet \\
rtr(sp, c, rp) \in cnet \land \\
CRtr[sp/sp?, rp/rp?, c/c?, msgs/msgs!, st/st!] \land \\
cnet' = (cnet \setminus \{rtr(sp, c, rp)\}) \cup msgs \land \\
pc' = pc \oplus \{st \mapsto pc\ st + 1\} \land \\
trace' = trace \frown \langle \mathsf{rtr}(sp, c) \rangle \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FMsg}} \\ \hline
\Delta FState \\ \hline
\exists\, rp, sp : PE;\ c : CHANNEL;\ m : MEM\_BLOCK;\ rt : THREAD \bullet \\
msg(rp, c, sp, m) \in cnet \land \\
CMsg[rp/rp?, sp/sp?, c/c?, m/m?, rt/rt!] \land \\
cnet' = cnet \setminus \{msg(rp, c, sp, m)\} \land \\
pc' = pc \oplus \{rt \mapsto pc\ rt + 1\} \land \\
trace' = trace \frown \langle \mathsf{msg}(rp, c) \rangle \\ \hline
\end{array}
$$

Bringing together the framing schemas for thread control and channel communication and using **Z**'s notation for flat schema definition, a computation step can now be defined as a *Receive* step or a *Schedule* step:

$$Receive \mathrel{\widehat{=}} FTerm\_msg \lor FAdvert \lor FFocus \lor FRtr \lor FMsg$$
$$Schedule \mathrel{\widehat{=}} FCreate \lor FSync \lor FTerminate \lor FSend \lor FReceive$$

4.3 Relating Design and Implementation

If the specification and correctness proofs about the specification are to give confidence in the safety of the implementation of the Testbed's operating system, then a convincing demonstration must be given that the specification is a true representation of the implementation. In this section I give just such a demonstration by showing how to relate each component of the specification to a part of the operating system.

The reader should note that the specification is assumed to operate with correct input. It is possible to extend specifications to deal with erroneous or inconsistent input but, as Hayes [4] reports, is not easy to model error recovery. Furthermore, when modelling a system as complex as an operating system to do so would obscure the real features of interest.

4.3.1 Basic entities

The *THREAD* and *CHANNEL* objects are implemented as records named `tcb` ('thread control block') and `ccb` ('channel control block'). The *COUNTER* objects are also realised as records, of the type `counter`. The *set* PE corresponds to the six processors of the Testbed and values from *PE* are represented by the natural numbers $1 \mathinner{\ldotp\ldotp} 6$. The *MEM_BLOCK* type is not represented directly because the specification does not represent the details of the Testbed's memory organisation.

The *CNETMSG* type is implemented as lists of records of predefined sizes. Certain fixed fields are used to mark the type of the message as *focusm*, *advert* or whatever and to contain the variable number of parameters associated with each message type. The *OP* type is not represented in the implementation because it is used only for constructing traces and the traces are there only to assist the proofs.

4.3.2 Functions and relations

Functions are represented by records and pointers or records and values. Consider for example, the *scontrol* function defined in *CState* with domain *PE* and range $CHANNEL \nrightarrow THREAD$. The range is represented in records of the type `ccb` by a field called `sctrl`. Each instance of a `ccb` record represents a particular channel and the value of `sctrl` is either `NULL`—representing a channel for which *scontrol* is undefined—or it is a pointer to a `tcb` record—representing a thread blocked on the channel. The fact that the domain of *scontrol* is the set *PE* is represented by giving each processor its own set of `ccb` records.

The *focus* function in *CState* shows the use of records and values. The range of *focus* is represented by a field of type `unsigned char` in the `ccb` records (although in practice the only values used are 0 meaning undefined or focus not set, and 1 meaning defined or focus set). The domain of *focus* is implemented, as before, by giving each processor its own set of `ccb` records.

Relations can be represented in the same way as functions or as sets of records. The *ready* relation between processors and threads is, for instance, implemented by giving each processor a pointer to a list of threads.

4.3.3 Initial states

The initial state *TInit* is implemented on the Testbed by having the operating system zero all fields in the `tcb` records before creating a new thread. The initialisation of *ready* occurs when the first thread of a new program is created. Similarly, the initial state *CInit* is implemented by zeroing the `ccb` records before use.

4.3.4 Operation schemas

The *TCreate* and *TTerminate* schemas are typical operation schemas so I shall use them to illustrate the correspondence between specification and implementation.

The $TCreate$ schema corresponds to the system function `trap6` which can be called by the user via the processor's exception mechanism. The input variable $pt?$ corresponds to the current value of $curr\_tcb$, a global pointer maintained by the operating system and pointing to the currently executing thread; $ct?$ is a parameter passed in a register by the calling thread; $p?$ is stored in a static variable which simply holds the number of the processor on which the operating system is running.

The $par\_bd\ ct? \neq p? \Rightarrow$ construction (in $TTerminate$) is implemented using the C construction `if (condition) block1 else block2` where the condition tests for set membership and the choice of the block to execute duplicates the implication. The definition of after state in terms of before state, e.g. $par\_bd' = par\_bd \cup \{ct? \mapsto p?\}$ (from the $TCreate$ schema) models assignment in C.

Other operation schemas use the $\lhd$ and $\setminus$ operators to remove maplets from sets. In the implementation, this corresponds to assigning a zero or `NULL` value. The **let** $pt == parent\ ct? \bullet$ construction (found in $TTerminate$, for example) corresponds to the declaration and initialisation of a local or automatic stack variable in a C function.

The framing schemas fall into two groups according to the way that they are 'driven'. Schemas $FCreate$, $FSync$, $FTerminate$, $FSend$ and $FReceive$ depend upon the current program instruction being of the appropriate type while the other framing schemas have preconditions such as $term(pp, pt) \in cnet$ and are applied, therefore, according to the messages that Centrenet is ready to deliver.

5 Specification of Testbed Migration

In this section I formally specify the non-standard features of the Testbed operating system that implement the mechanism for task migration. Section 6, which follows, contains the proofs of correctness based on the specification presented here and in the preceding section. It is interesting to note that, given the specification of threads and channels in Section 4, the additional specification required for migration is relatively short.

I model the two operations of disconnecting a thread and reconnecting a thread. The thread migration protocol takes care of moving the thread register context and control information and updating the channel control blocks. The protocol which enables recently migrated threads to retrieve their code and data memory pages is not modelled in this report because the memory page protocol is independent of the thread migration protocol and to present it here would add little of new interest.

5.1 Migration Protocol

This section gives, by means of a simple example, an informal introduction to the migration protocol implemented in the Testbed operating system.

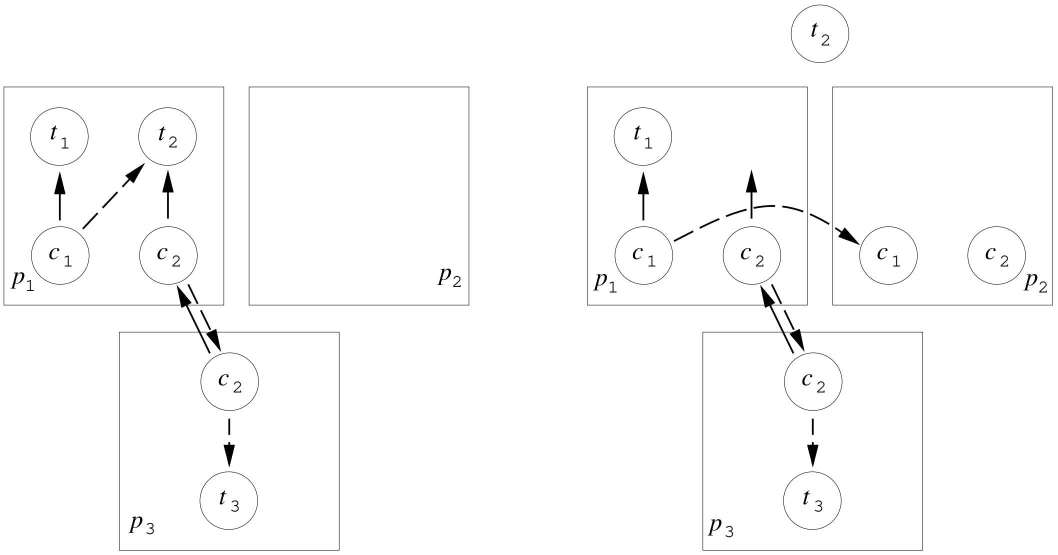

Figure 8: *Example thread migration: initial state and state after thread disconnection.*

With reference to the left half of Figure 8, consider a scenario involving three processors p_1, p_2 and p_3 and three threads t_1, t_2 and t_3. Two threads are executing on one of the processors, one on another and the third processor is idle. There are two communication channels in use: a local channel between t_1 and t_2 implemented by the shared channel control block (CCB) labelled c_1 on processor p_1 and a remote channel between t_2 and t_3 implemented by two CCBs on p_1 and p_3 respectively. Each CCB has two outgoing arcs, the dashed arc represents the *rloc* value and points to the last known location for the receiver and the undashed arc represents the *sloc* value and points to the last known location for the sender. For example, the CCB c_2 on processor p_1 thinks its sender is on p_1 and its receiver is on p_3.

A possible optimisation would be to migrate thread t_2 from processor p_1 to p_2. This removes the competition between t_1 and t_2 for CPU cycles on p_1 but it also changes the local (low latency) channel between t_1 and t_2 into a remote (high latency) channel. Whether the new configuration is more efficient or not is program dependent, i.e. it depends on the relative amounts of computation and communication that t_1 and t_2 perform.

The first phase of thread migration is modelled by the *MDisconnect* schema (presented in the next section) and involves disconnecting thread t_2 from its environment on processor p_1. The CCBs used by t_2 are examined, a snapshot is taken of their current status and they are updated to reflect the fact that t_2 is soon going to be at p_2. Then, the CCB snapshot is put into a Centrenet message along with t_2's register context (and some other control information) and the message queued for transmission to p_2.

The right half of Figure 8 shows the situation at this point. Thread t_2 is somewhere in transit between processors p_1 and p_2, CCB c_1 has been updated

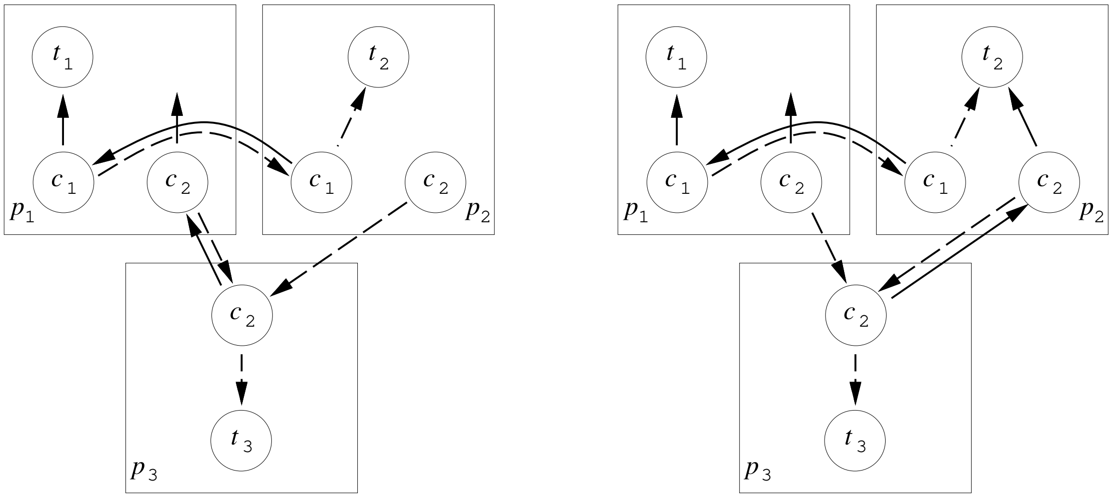

Figure 9: *Example thread migration: state after thread reconnection and final state.*

to have the (soon-to-be) correct location for its receiver t_2 and the other CCBs are unchanged.

Phase two of the migration occurs when the Centrenet message containing t_2 and its associated information arrives at p_2 and begins to be processed. As modelled by the *MConnect* schema, t_2 is unpacked from the message and reconnected in its new environment. The channel status from the CCB snapshot is integrated with the local CCBs and the thread is put into the processor ready queue.

The new system state is shown in the left half of Figure 9 — thread t_2 is in place and its local CCBs have been updated to locate the sender for channel 1 on processor p_1 and the receiver for channel 2 on p_3.

Although the CCBs for channel 2 on p_1 and p_2 are not updated with respect to their sender locations, this is done by the next communication on channel 2: the *CSend* operation updates the sender location, *sloc*, on p_2 and the reception of the *focusm* message at p_3 in the *CFocus* operation will update *sloc* at p_3 — as shown in the right half of Figure 9. In general, it is the sender who will initiate communication and so it is important that the sender have an up-to-date idea of where the receiver is, but it is not necessary for the receiver to remember where the sender is until communication is under way.

5.2 Migration Operations

The *MDisconnect* schema models the disconnecting of a 'migrating thread' $mt?$ from its environment on the 'from processor' $fp?$ and the submission of $mt?$'s register context and channel snapshot to Centrenet for transmission to the 'to processor' $tp?$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \ MDisconnect\ \ }} \\
\Delta FState \\
mt? : THREAD \\
fp?, tp? : PE \\
\hline
(fp?, mt?) \in ready \\
mt? \in \mathrm{dom}(cntr) \Rightarrow first(cntr\ mt?) = 0 \\
(\mathbf{let}\ sc == \{c : CHANNEL \mid mt? = sender\ fp?\ c\}; \\
rc == \{c : CHANNEL \mid mt? = receiver\ fp?\ c\} \bullet \\
(\mathbf{let}\ scl == \{c : sc \bullet (c, rloc\ fp?\ c)\}; \\
rcl == \{c : rc \bullet (c, sloc\ fp?\ c, \mathbf{if}\ fp? \mapsto c \in focus\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\} \bullet \\
focus' = focus \setminus \{c : rc \bullet fp? \mapsto c\} \wedge \\
rloc' = rloc \oplus \{c : rc \bullet fp? \mapsto \{c \mapsto tp?\}\} \wedge \\
cnet' = cnet \cup \{thrd(tp?, mt?, scl, rcl)\})) \\
\hline
\end{array}
$$

The schema has the precondition that the migrating thread must be in the ready queue and not, for instance, blocked on channel communication. It is deemed too complicated to migrate threads that are part-way through communication or some other activity. If the precondition holds then the two temporary variables sc ('sending channels') and rc ('receiving channels') are defined holding, respectively, all the channels which $mt?$ has sent and received over. The sending channels are then used to construct the 'sending channels list' scl which gives for each sending channel the last known location for its receiver.

The receiving channels are used to construct the 'receiving channels list' rcl which gives for each receiving channel the last known location of the sender and an indication of whether the channel focus is present. The focus for all receiving channels is forced to undefined. What this is doing, in effect, is to migrate the foci which are present with the thread. Finally, the receiver location is updated to point to the new site and the migration message is queued for transmission by Centrenet.

The *MConnect* schema models the (re)connecting of a thread mt at the destination site $tp?$. This involves moving mt from the Centrenet message and putting it in the ready queue and integrating the channel snapshot with the information already present in the CCBs.

The precondition for the schema is that a message of the appropriate form is found in the set $cnet$. If the precondition holds, then the state component $rloc$ is updated so that for all mt's sending channels $rloc$ now points to the processor specified in scl and for all mt's receiving channels $rloc$ points to mt's new location, $tp?$.

The state component $sloc$ is also updated for all mt's receiving channels to point to the processor p defined in rcl. The foci migrated with the thread are unpacked and attached to the appropriate channel by $focus' = focus \cup \{c : CHANNEL; p : PE \mid (c, p, 1) \in rcl \bullet tp? \mapsto c\}$ and the used flag is set for all mt's receiving

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MConnect \quad}} \\
\Delta FState \\
tp? : PE \\
\hline
\exists\, mt : THREAD;\ scl : \mathbb{P}(CHANNEL \times PE);\ rcl : \mathbb{P}(CHANNEL \times PE \times \mathbb{N}) \bullet \\
thrd(tp?, mt, scl, rcl) \in cnet \land \\
rloc' = rloc \oplus \\
\qquad \{c : CHANNEL;\ p : PE \mid (c, p) \in scl \bullet tp? \mapsto \{c \mapsto p\}\} \oplus \\
\qquad \{c : CHANNEL;\ p : PE;\ n : \mathbb{N} \mid (c, p, n) \in rcl \bullet tp? \mapsto \{c \mapsto tp?\}\} \land \\
sloc' = sloc \oplus \{c : CHANNEL;\ p : PE \mid (c, p, 1) \in rcl \bullet tp? \mapsto \{c \mapsto p\}\} \land \\
focus' = focus \cup \{c : CHANNEL;\ p : PE \mid (c, p, 1) \in rcl \bullet tp? \mapsto c\} \land \\
used' = used \cup \{c : CHANNEL;\ p : PE;\ n : \mathbb{N} \mid (c, p, n) \in rcl \bullet tp? \mapsto c\} \land \\
sender' = sender \oplus \\
\qquad \{c : CHANNEL;\ p : PE \mid (c, p) \in scl \bullet tp? \mapsto \{c \mapsto mt\}\} \land \\
receiver' = receiver \oplus \\
\qquad \{c : CHANNEL;\ p : PE;\ n : \mathbb{N} \mid (c, p, n) \in rcl \bullet tp? \mapsto \{c \mapsto mt\}\} \\
\hline
\end{array}
$$

channels. Finally, *sender* is updated to record the channels that *mt* sends on and *receiver* to record the channels *mt* receives on.

6 Verification of Specification

This section has three subsections and three proofs. The first two proofs assume that thread migration is not allowed and show that the Testbed implements the occam semantics for, respectively, synchronisation between parent and children threads and synchronisation between sender and receiver threads during channel communication. The third proof shows that the semantics for channel communication are preserved even when one of the threads using a channel undergoes migration between consecutive communications.

I do not provide any proofs about the unidirectional and point-to-point properties of Testbed channels because they are best enforced by the programmer and/or compiler. For practical reasons, the operating system does implement some checking of channels but, because it is very inefficient to implement complete checking, the specification and proofs use an assumption that user programs never attempt to violate these properties.

6.1 Thread Synchronisation

This section begins with three assumptions and a lemma which rule out certain kinds of erroneous behaviour on the part of threads. The proof that the Testbed implements correct synchronisation between parent threads and children then proceeds by induction on the number of synchronisations a given parent thread has undergone.

6.1.1 Assumptions and lemmas

There are a number of conditions which are necessary for the correct functioning of the Testbed but which cannot be derived from the specification. The first two assumptions relate to the proper operation of the compiler and the third assumption to the expected operation of the Centrenet communication system. The lemma demonstrates that the specification meets the intended semantics for functions *parent* and *par_bd* and is stated here to simplify the synchronisation proof.

Assumption 1 A given thread may not be created more than once: it is not possible for the $create(ct_1)$ instruction to belong to two different programs or to occur at two different places in the same program.

$$\begin{array}{l}
\forall\, FState;\ pt_1, pt_2, ct_1 : THREAD;\ n_1, n_2 : \mathbb{N} \bullet \\
program\, pt_1\, n_1 = create(ct_1) \land \\
\quad program\, pt_2\, n_2 = create(ct_1) \Rightarrow \\
\qquad pt_1 = pt_2 \land n_1 = n_2
\end{array}$$

Assumption 2 All programs have a terminate instruction and that instruction is always the last instruction in the program. All synchronise instructions are preceded by at least one create instruction. All pairs of synchronise instructions have at least one intervening create instruction. (These constraints should be satisfied by any competent compiler.)

$$\begin{array}{l}
\forall\, FState \bullet \\
\forall\, p : \mathrm{ran}(program) \bullet \\
(\exists_1\, n : \mathbb{N} \bullet p\ n = terminate \land n = \#p) \land \\
(\forall\, n_1 : \mathbb{N} \bullet p\ n_1 = sync \Rightarrow \\
\quad (\exists\, n_2 : 1 \mathrel{..} n_1;\ ct : THREAD \bullet p\ n_2 = create(ct))) \land \\
(\forall\, n_1, n_2 : \mathbb{N} \bullet p\ n_1 = sync \land p\ n_2 = sync \land n_2 > n_1 \Rightarrow \\
\quad (\exists\, n_3 : n_1 \mathrel{..} n_2;\ ct : THREAD \bullet p\ n_3 = create(ct)))
\end{array}$$

Assumption 3 Centrenet does not introduce spurious messages, or duplicate or lose messages. Additionally, Centrenet will always deliver messages within a finite time. The first statement need not be represented as it is a direct consequence of the use of the set *cnet* to model Centrenet. The second statement cannot be stated in terms of the specification because I have not formally developed any notion of time.

Lemma 1 Where the functions *parent* and *par_bd* are defined for a child thread *ct*, *parent ct* always gives the thread that created *ct* and *par_bd ct* always gives the processor on which the parent resides.

Proof of lemma I note that only *TCreate* and *TTerminate* change *parent*, the former associates the parent thread with the child upon creation of that child and the latter removes any mappings from the child upon termination of that child. Similarly, only *TCreate* and *TTerminate* modify *par_bd*, the former associates the parent's processor with the child upon creation of that child and the latter removes any mappings from the child upon its termination. The migration schema *MDisconnect* prohibits parents with children from migrating so the child/processor association remains constant during the child's lifetime.

6.1.2 Synchronisation proof

I prove that when a thread requests to be suspended on completion of its children it will be removed from the ready queue at least until all of its children have terminated. The proof proceeds by induction: the base case considers all state transitions from the creation of the parent thread up to the unblocking of the parent after its first synchronisation and the step case considers an arbitrary, subsequent synchronisation.

Note that for concision the entire proof is universally quantified over FState and threads pt and ct.

Synchronisation proof base case:

$$(\exists_1\, n : \mathbb{N} \bullet trace\ n = \mathsf{sync}(pt)) \land last\ trace = \mathsf{create}(pt, ct) \Rightarrow$$
$$(pt \in \operatorname{ran} parent \Rightarrow pt \notin \operatorname{ran} ready)$$

The base case can be stated informally as follows: after the first synchronisation request and before the creation of another child, if the parent still has children then it cannot be in the ready queue.

$$last\ trace = \mathsf{create}(pt, ct) \Rightarrow ct \notin \operatorname{dom} cntr$$

The first step in the argument is to show that threads begin with no counter at all. When the first thread of the program is defined by *TInit*, only the null thread has a counter and for each new thread *ct*, *TCreate* ensures *ct* has no counter.

$$last\ trace \in \{\mathsf{sync}(pt), \mathsf{terminate}(ct), \mathsf{term\_msg}(ct)\} \Rightarrow \mathsf{create}(pt, ct) \in \operatorname{ran}(trace)$$

The first counter-modifying operation to befall the parent thread must be its first application of *FCreate*. None of the other counter-modifying operations are possible: *FSync* cannot occur until *after* the first *FCreate* by Assumption 2; *FTerminate* only modifies the counter belonging to the parent of the terminating thread as defined by *parent ct* and our thread does not participate in the *parent* function until *after* its first *FCreate*; likewise *FTerm_msg* only modifies the counter of the thread specified in the *term* message, the *term* message must have been generated by a preceding *FTerminate* since Assumption 3 prohibits Centrenet from inventing messages, and that preceding *FTerminate* again uses *parent* to define the thread in the *term* message.

$$\mathit{last\ trace} = \mathsf{create}(pt, ct) \Rightarrow \mathit{cntr}\ pt = (1, \mathit{null\_thread}) \wedge pt = \mathit{parent}\ ct$$

Now that I have argued that threads begin with no counter and the first counter-modifying operation they perform is *FCreate*, I conclude that the first time a thread becomes a parent it receives a counter with a value of $(1, \mathit{null\_thread})$.

$$\begin{aligned} &\mathsf{create}(pt, ct) \in \operatorname{ran} \mathit{trace} \wedge \mathsf{sync}(pt) \notin \operatorname{ran} \mathit{trace} \Rightarrow \\ &\quad \mathit{cntr}\ pt = (\#\{t : \mathit{THREAD} \mid pt = \mathit{parent}\ t\} + \\ &\qquad \#\{p : \mathit{PE} \mid \mathit{term}(p, pt) \in \mathit{cnet} \bullet \mathit{term}(p, pt)\}, \mathit{null\_thread}) \end{aligned}$$

Between creating the first thread and requesting to synchronise, our thread has a counter of value $(n, \mathit{null\_thread})$ where n is the number of children yet to terminate plus the number of relevant *term* messages in Centrenet. To support this statement, consider all the counter-modifying operations which may be applied between the first thread creation and synchronisation:

FCreate increments the counter because the predicate $pt? \in \operatorname{dom}(\mathit{cntr})$ is true. The schema also creates a thread so the first part of the counter still correctly represents the number of threads in the system plus the relevant *term* messages.

FTerminate either terminates the child and decrements the counter, or terminates the child, adds a *term* message to Centrenet and does not change the counter. In both cases the first part of the counter reflects the number of children plus relevant *term* messages and the second part of the counter is assigned the *null_thread* or retains its value.

FTerm_msg consumes a *term* message and decrements the counter to give it the value $(0, \mathit{null\_thread})$ or $(\mathit{first}\ c - 1, \mathit{second}\ c)$—in both cases the first part of the counter reflects the number of children plus relevant *term* messages and the second part of the counter is assigned the *null_thread* or retains its value.

Hence, I conclude that between creating the first thread and requesting to synchronise our thread has a counter of the appropriate value.

$$\begin{aligned} &\mathit{last\ trace} = \mathsf{sync}(pt) \Rightarrow \\ &\quad (\mathit{first}(\mathit{cntr}\ pt) = 0 \wedge pt \in \operatorname{ran} \mathit{ready}) \vee \\ &\qquad (\mathit{second}(\mathit{cntr}\ pt) = pt \wedge pt \notin \operatorname{ran} \mathit{ready}) \end{aligned}$$

When our parent thread requests to synchronise it will be returned to the ready queue immediately if there are no child threads still active or relevant *term* messages; otherwise it will be suspended. From the argument above, the value of our thread's counter immediately prior to the application of *FSync* is the number of children plus the relevant *term* messages. From inspection of *FSync* it is obvious that if there are no children and no relevant *term* messages, i.e. $\mathit{first}(\mathit{cntr}\ pt?) = 0$, then no action is taken, the counter is unchanged and our parent remains in the ready queue. Alternatively, if there are children and/or relevant *term* messages then the predicate $\mathit{first}(\mathit{cntr}\ pt?) > 1$ will be true and the counter will be updated by setting the thread value to our parent $pt?$ and our parent will be deleted from the ready queue. I note that if our parent is not suspended then its counter is not changed and has the value $(0, \mathit{null\_thread})$.

$$\mathit{cntr}\ pt = (n, pt) \Rightarrow n = \#\{t : \mathit{THREAD} \mid pt = \mathit{parent}\ t\} + \\ \#\{p : \mathit{PE} \mid \mathit{term}(p, pt) \in \mathit{cnet} \bullet \mathit{term}(p, pt)\}$$

If our parent thread is blocked on synchronisation then each time one of its children terminates our parent's counter will be decremented or an appropriate *term* message produced. Additionally, our parent's counter will also be decremented each time an appropriate *term* message is consumed. Consider an application of *FTerminate* when one of our parent's children terminates: if the child resides on the same processor ($\mathit{par\_bd}\ ct = p?$) and it is the last child of the parent ($\mathit{first}\ c = 1$) then the counter is reduced to zero, if it is not the last child then the counter is decremented to obtain the value $(\mathit{first}\ c - 1, \mathit{second}\ c)$. If the child does not reside on the same processor then a *term* message is submitted for transmission by Centrenet specifying the parent thread and its processor board (by Lemma 1 this *term* message is directed at the appropriate parent at the appropriate processor). Consider an application of *FTerm_msg* and the consumption of a *term* message directed at our parent: if the present counter has a numeric value of 1 then the counter is updated to $(0, \mathit{null\_thread})$ and if the present numeric value is greater than 1 then the counter is updated to $(\mathit{first}\ c - 1, \mathit{second}\ c)$. In both cases the numeric part of the counter is decremented by 1.

When a counter with a thread blocked on it is decremented to zero the thread is returned to the ready queue. This is justified by examination of *TTerminate* and *TTerm_msg*. When *TTerminate* decrements a counter of value $(1, pt)$ it updates *ready* to the value $(\mathit{ready} \setminus \{p? \mapsto ct?\}) \oplus \{p? \mapsto pt\}$. When *TTerm_msg* decrements a counter of value $(1, pt)$ it updates *ready* to the value $\mathit{ready} \oplus \{p? \mapsto pt\}$. Note that in both cases our parent's counter is set to $(0, \mathit{null\_thread})$.

This completes the base case of the proof of the correctness of thread synchronisation.

Synchronisation proof step case: I noted for the base case that the parent's counter is set to $(0, \mathit{null\_thread})$ when the parent is restarted after being blocked and I also noted that if the parent requests synchronisation, but is not blocked because there are no active children or relevant *term* messages, then its counter must also have the value $(0, \mathit{null\_thread})$. This counter value may therefore be assumed as the initial counter value for the step case.

The same arguments that were given for the base case now apply to the step case except I note that in the first application of *FCreate* the predicate $pt? \in \mathrm{dom}(cntr)$ is true and our parent's counter is assigned the value $(\mathit{first}(cntr\ pt?) + 1, \mathit{null\_thread})$. Since I have just shown that the initial counter value may be assumed to be $(0, \mathit{null\_thread})$ the value of the counter assigned by the first application of *FCreate* is therefore $(1, \mathit{null\_thread})$, exactly as argued for the base case. This completes the argument for the step case. □

Corollary Terminating children executing on the same processor as their parent decrement the parent's counter as part of the *TTerminate* operation. Terminating children not sharing a processor with their parent cause a *term* message to be submitted to Centrenet as part of the *TTerminate* operation. By Assumption 3 this message will be delivered within a finite amount of time to the parent's processor where the *TTerm_msg* operation will cause the parent's counter to be decremented. Hence, the suspended parent will be returned to the ready queue within a finite time from the termination of the last child.

The theorem and corollary complete the proof that the Testbed implements the occam semantics for thread control.

6.2 Channel Synchronisation

This subsection is in two parts. An assumption is stated which constrains the sequences of channel operations allowed and a series of lemmas are presented which give the detailed working of the proof. The inductive proof for correct channel synchronisation is based on the lemmas and is given in the latter half of the subsection.

6.2.1 Assumptions and lemmas

It is, unfortunately, beyond the scope of this report to specify the behaviour of the Testbed channel communication protocol when subjected to sequences of communication requests contravening the occam channel semantics. The restrictions on the sequences of communication requests required here are as follows:

$$
\begin{array}{l}
\forall\, FState;\ t_1 : THREAD;\ c : CHANNEL;\ m_1, m_2 : MEM\_BLOCK \bullet \\
\quad send(c, m_1) \in \mathrm{ran}(program\ t_1) \Rightarrow receive(c, m_2) \notin \mathrm{ran}(program\ t_1) \wedge \\
\quad (\forall\, t_2 : THREAD \setminus \{t_1\} \bullet \\
\qquad send(c, m_1) \in \mathrm{ran}(program\ t_1) \Rightarrow send(c, m_2) \notin \mathrm{ran}(program\ t_2) \wedge \\
\qquad receive(c, m_1) \in \mathrm{ran}(program\ t_1) \Rightarrow receive(c, m_2) \notin \mathrm{ran}(program\ t_2))
\end{array}
$$

Assumption 4 If a thread t_1 sends on a channel c then it cannot also receive on c. If t_1 sends on c then no other thread may send on the same channel, and if t_1 receives on c then no other thread may receive on the same channel.

Lemmas 2 to 20 all have the same format. A new schema Opn is defined in terms of one or more channel operations applied to a previously defined state, e.g. $Op4$ is the request-to-receive operation applied to the state which models a channel that has never been used. Next, a state schema Sn is defined in which the relevant consequences of Opn are shown, e.g. $S4$ states that $Op4$ causes the receiver to be blocked on the channel.

The Sn state schemas all take the same input and output arguments, so the arguments have been packaged up into a schema called $Args$. Informally, $c?$ is the channel being communicated over, $st?$ and $rt?$ are the sender and receiver threads, $sp?$ and $rp?$ the processors on which the sender and receiver are executing and $msgs!$ contains any tuples to be submitted to Centrenet for transmission.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Args}} \\
\hline
c? : \mathit{CHANNEL} \\
st?, rt? : \mathit{THREAD} \\
sp?, rp? : \mathit{PE} \\
msgs! : \mathbb{P}\,\mathit{CNETMSG} \\
\hline
\end{array}
$$

As the lemmas will ultimately be part of a proof-by-induction, they are divided into those which relate to the proof base case and those which relate to the step case. The initial state for the base case is characterised by $SBase$. This initial state is a subset of $FInit$ and it shows that there is no sender or receiver, at any processor, blocked on the channel of interest and that there is no focus set.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SBase}} \\
\hline
\mathit{FState}; \mathit{Args} \\
\hline
\forall p : \mathit{PE} \bullet c? \notin \mathrm{dom}(\mathit{scontrol}\, p) \land c? \notin \mathrm{dom}(\mathit{rcontrol}\, p) \land c? \notin \mathrm{dom}(\mathit{rloc}\, p) \\
c? \notin \mathrm{ran}\, \mathit{focus} \\
\hline
\end{array}
$$

The state *after* the first (and any subsequent) communication has completed, is characterised by $SStep$: no sender or receiver is blocked on the channel at any processor, the channel focus is not set anywhere, the channel is marked as used at the receiver's processor, the receiver location is known at both the sender's and receiver's processors to be $rp?$ and the sender and receiver identifications are stored at the sender's and receiver's processors respectively.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SStep}} \\
\hline
\mathit{FState}; \mathit{Args} \\
\hline
\forall p : \mathit{PE} \bullet c? \notin \mathrm{dom}(\mathit{scontrol}\, p) \land c? \notin \mathrm{dom}(\mathit{rcontrol}\, p) \\
c? \notin \mathrm{ran}\, \mathit{focus} \land (rp?, c?) \in \mathit{used} \\
\mathit{rloc}\, sp?\, c? = rp? \land \mathit{rloc}\, rp?\, c? = rp? \\
(c?, st?) \in \mathit{sender}\, sp? \land (c?, rt?) \in \mathit{receiver}\, rp? \\
\hline
\end{array}
$$

The base case lemmas

Lemma 2 The $Op2$ operation models a sender thread $st?$ executing on $sp?$ requesting to send over a channel $c?$. The previously defined state $SBase$ implies that this is the first-ever use of the channel. The $S2$ schema shows the important consequences of applying $Op2$.

$Op2 \mathrel{\widehat=} SBase \mathbin{;} CSend$

$$\begin{array}{|l}
\multicolumn{1}{l}{S2} \\
\hline
\Delta FState; Args \\
\hline
Op2 \Rightarrow st? \notin \operatorname{ran} ready' \land \\
\qquad (\exists_1 p : PE \bullet c? \in \operatorname{dom}(scontrol'\,p)) \land (c?, st?) \in scontrol'\,sp? \land \\
\qquad (\forall p : PE \bullet c? \notin \operatorname{dom}(rcontrol'\,p)) \land \\
\qquad (\exists_1 p : PE \bullet (p, c?) \in focus' \land p = sp?) \land \\
\qquad (rp?, c?) \notin used' \land \\
\qquad (c?, st?) \in sender'\,sp? \land \\
\qquad (\forall p : PE \setminus \{sp?\} \bullet advert(p, c?, sp?) \in msgs!) \\
\hline
\end{array}$$

The sender is removed from the ready queue ($st? \notin \operatorname{ran} ready'$), the sender is blocked on the channel at $sp?$ and only at $sp?$ ($(\exists_1 p : PE \bullet c? \in \operatorname{dom}(scontrol'\,p)) \land (c?, st?) \in scontrol'\,sp?$), no receiver thread is blocked anywhere on the channel ($\forall p : PE \bullet c? \notin \operatorname{dom}(rcontrol'\,p)$), the channel focus is set at $sp?$ and only at $sp?$ ($\exists_1 p : PE \bullet (p, c?) \in focus' \land p = sp?$), the channel-used flag is not set at any processor ($(rp?, c?) \notin used'$), the sender thread is recorded as the legal sender for the channel at $sp?$ ($(c?, st?) \in sender'\,sp?$) and *advert* messages are queued for transmission by Centrenet to all processors other than $sp?$ ($\forall p : PE \setminus \{sp?\} \bullet advert(p, c?, sp?) \in msgs!$).

The functions *ready*, *scontrol*, *focus* and *sender* are updated by *CSend* as is the output argument $msgs!$, note that *SBase* implies that there is no waiting receiver and that $rloc\,sp?\,c?$ is not defined. Functions *rcontrol* and *used* are carried over from *SBase*.

Lemma 3 The schema $S3$ shows what happens when the first-ever receive request on a channel $c?$ finds a sender already waiting.

$Op3 \mathrel{\widehat=} S2 \mathbin{;} CReceive$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad S3 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta FState;\ Args \\
\hline
sp? = rp? \land Op3 \Rightarrow st? \in \operatorname{ran} ready' \land \\
\qquad (\forall p : PE \bullet c? \notin \operatorname{dom}(scontrol'\ p) \land c? \notin \operatorname{dom}(rcontrol'\ p)) \land \\
\qquad c? \notin \operatorname{ran} focus' \land (rp?, c?) \in used' \land \\
\qquad rloc'\ sp?\ c? = rp? \land rloc'\ rp?\ c? = rp? \land \\
\qquad (c?, st?) \in sender'\ sp? \land (c?, rt?) \in receiver'\ rp? \\
\hline
\end{array}
$$

The sender is unblocked from the channel and returned to the ready queue, the receiver consumes the channel focus and sets the channel used flag, the receiver location is updated at $rp?$ and therefore at $sp?$ since they are the same processor, finally the receiver identification ($receiver$) is set—the sender identification carries over from $S2$.

Lemma 4 The $S4$ schema models the first-ever receive request on a channel $c?$: the receiver is removed from the ready queue and blocked on the channel, the receiver location is set at $rp?$ (and therefore $sp?$) and the receiver identification is recorded. The values of $scontrol$ and $focus$ carry over from $SBase$.

$Op4 \mathrel{\widehat{=}} SBase \fatsemi CReceive$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad S4 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta FState;\ Args \\
\hline
sp? = rp? \land Op4 \Rightarrow rt? \notin \operatorname{ran} ready' \land \\
\qquad (\forall p : PE \bullet c? \notin \operatorname{dom}(scontrol'\ p)) \land \\
\qquad (\exists_1\ p : PE \bullet c? \in \operatorname{dom}(rcontrol'\ p)) \land (c?, rt?) \in rcontrol'\ rp? \land \\
\qquad c? \notin \operatorname{ran} focus' \land \\
\qquad rloc'\ sp?\ c? = rp? \land rloc'\ rp?\ c? = rp? \land \\
\qquad (c?, rt?) \in receiver'\ rp? \\
\hline
\end{array}
$$

Lemma 5 The $S5$ schema gives the implications of a send after the first-ever receive on the same processor and for a channel $c?$: the receiver is unblocked from the channel and returned to the ready queue thus $scontrol$ and $rcontrol$ are completely undefined for $c?$, the channel used flag is set and the sender identification is set. The values of $focus$ and $rloc$ carry over from $S4$.

$Op5 \mathrel{\widehat{=}} S4 \fatsemi CSend$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad S5 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta FState; Args \\
\hline
sp? = rp? \land Op5 \Rightarrow rt? \in \operatorname{ran} ready' \land \\
\quad (\forall p : PE \bullet c? \notin \operatorname{dom}(scontrol'\,p) \land c? \notin \operatorname{dom}(rcontrol'\,p)) \land \\
\quad c? \notin \operatorname{ran} focus' \land (rp?, c?) \in used' \land \\
\quad rloc'\,sp?\,c? = rp? \land rloc'\,rp?\,c? = rp? \land \\
\quad (c?, st?) \in sender'\,sp? \land (c?, rt?) \in receiver'\,rp? \\
\hline
\end{array}
$$

Lemma 6 The schema $S6$ shows what happens when an *advert* message, emitted by the first-ever send on channel $c?$ in $S2$, arrives at another processor: the sender location for the channel is set. All the other functions carry their values over from $S2$.

$Op6 \mathrel{\widehat=} S2 \mathbin{⨾} CAdvert$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad S6 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta FState; Args \\
\hline
sp? \neq rp? \land Op6 \Rightarrow (c?, sp?) \in sloc'\,rp? \land \\
\quad (\exists_1 p : PE \bullet c? \in \operatorname{dom}(scontrol'\,p) \lor (p, c?) \in focus' \land p = sp?) \land \\
\quad (\forall p : PE \bullet c? \notin \operatorname{dom}(rcontrol'\,p)) \land \\
\quad (rp?, c?) \notin used' \land \\
\quad (c?, st?) \in sender'\,sp? \\
\hline
\end{array}
$$

Lemma 7 The $S7$ schema models the first-ever receive operation on a channel $c?$ for which an *advert* message has already been received: the receiver is removed from the ready queue and blocked on the channel, the receiver location is set at $rp?$, the receiver identification is set at $rp?$ and an *rtr* message is queued for transmission back to the sender. The values of *scontrol*, *focus* and *sender* carry over from $S6$.

$Op7 \mathrel{\widehat=} S6 \mathbin{⨾} CReceive$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad S7 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta FState; Args \\
\hline
sp? \neq rp? \land Op7 \Rightarrow rt? \notin \operatorname{ran} ready' \land \\
\quad (\exists_1 p : PE \bullet c? \in \operatorname{dom}(scontrol'\,p)) \land scontrol'\,sp?\,c? = st? \land \\
\quad (\exists_1 p : PE \bullet c? \in \operatorname{dom}(rcontrol'\,p)) \land rt? = rcontrol'\,rp?\,c? \land \\
\quad (\exists_1 p : PE \bullet (p, c?) \in focus' \land p = sp?) \land \\
\quad rloc'\,rp?\,c? = rp? \land \\
\quad (c?, st?) \in sender'\,sp? \land (c?, rt?) \in receiver'\,rp? \land \\
\quad rtr(sloc\,rp?\,c?, c?, rp?) \in msgs! \\
\hline
\end{array}
$$

Lemma 8 The schema $S8$ shows what happens when an *advert* message, emitted by a sender in $S2$ during the first-ever communication over $c?$, is processed at a site where there is a waiting receiver: an *rtr* message is returned to the sender's processor. The values of *scontrol*, *focus* and *sender* carry over from the application of *CSend* and the values of *rcontrol*, *rloc* and *receiver* carry over from the application of *CReceive*.

$Op8 \mathrel{\widehat=} (CSend \mathbin{;} CReceive \mathbin{;} CAdvert) \lor (CReceive \mathbin{;} CSend \mathbin{;} CAdvert)$

S8

$\Delta FState; Args$

$$\begin{aligned}
& S2 \land Op8 \Rightarrow (\exists_1\, p : PE \bullet c? \in \operatorname{dom}(scontrol'\, p)) \land scontrol'\, sp?\, c? = st? \land \\
& \quad (\exists_1\, p : PE \bullet c? \in \operatorname{dom}(rcontrol'\, p)) \land rt? = rcontrol'\, rp?\, c? \land \\
& \quad (\exists_1\, p : PE \bullet (p, c?) \in focus' \land p = sp?) \land rloc'\, rp?\, c? = rp? \land \\
& \quad (c?, st?) \in sender'\, sp? \land (c?, rt?) \in receiver'\, rp? \land \\
& \quad rtr(sloc\, rp?\, c?, c?, rp?) \in msgs!
\end{aligned}$$

Lemma 9 The $S9$ schema models an *rtr* message being processed at the sender's processor during the first-ever communication over $c?$.

$Op9 \mathrel{\widehat=} (S7 \lor S8) \mathbin{;} CRtr$

S9

$\Delta FState; Args$
$st! : THREAD$

$$\begin{aligned}
& sp? \neq rp? \land Op9 \Rightarrow st? = st! = scontrol'\, sp?\, c? \land st? \in \operatorname{ran} ready' \land \\
& \quad (\forall\, p : PE \bullet c? \notin \operatorname{dom}(scontrol'\, p)) \land \\
& \quad (\exists_1\, p : PE \bullet c? \in \operatorname{dom}(rcontrol'\, p)) \land rt? = rcontrol'\, rp?\, c? \land \\
& \quad c? \notin \operatorname{ran} focus' \land \\
& \quad rloc'\, sp?\, c? = rp? \land rloc'\, rp?\, c? = rp? \land \\
& \quad (c?, st?) \in sender'\, sp? \land (c?, rt?) \in receiver'\, rp? \land \\
& \quad msg(rp?, c?, sp?, buffer\, sp?\, c?) \in msgs!
\end{aligned}$$

The sender is unblocked from the channel and returned to the ready queue, the focus is consumed on the receiver's behalf, the receiver location is set at $sp?$ and a data message is queued for transmission to the receiver's processor. The values of *rcontrol*, *rloc rp?*, *sender* and *receiver* carry over from $S7$ or $S8$.

Lemma 10 The $S10$ schema gives the implications of a data message being processed at a receiver's processor during the first-ever communication over $c?$: the receiver is unblocked from the channel (so $c?$ is not defined anywhere in *rcontrol*) and returned to the ready queue and the channel used flag is set. The values of *scontrol*, *focus*, *sender* and *receiver* carry over from $S9$.

$Op10 \mathrel{\widehat=} S9 \mathbin{;} CMsg$

$$
\begin{array}{|l}
\multicolumn{1}{l}{S10} \\
\hline
\Delta FState; Args \\
st!, rt! : THREAD \\
\hline
sp? \neq rp? \land Op10 \Rightarrow rt? = rt! = rcontrol'\, rp?\, c? \land \\
\quad rt? \in \operatorname{ran} ready' \land \\
\quad (\forall p : PE \bullet c? \notin \operatorname{dom}(scontrol'\, p) \land c? \notin \operatorname{dom}(rcontrol'\, p)) \land \\
\quad c? \notin \operatorname{ran} focus' \land (rp?, c?) \in used' \land \\
\quad rloc'\, sp?\, c? = rp? \land rloc'\, rp?\, c? = rp? \land \\
\quad (c?, st?) \in sender'\, sp? \land (c?, rt?) \in receiver'\, rp? \\
\hline
\end{array}
$$

The step case lemmas

Lemma 11 The schema $S11$ shows what happens when a send request is made for a channel that has been used before and is therefore known to be local and to have no waiting receiver: the sender is removed from the ready queue and blocked on the channel (so there is now exactly one sender blocked on the channel), the channel focus is set and the sender identification is overwritten but, by Assumption 4, the new value must be the same as the old value. The functions *rcontrol*, *rloc* and *receiver* are carried over from *SStep*.

$Op11 \mathrel{\widehat=} SStep \mathbin{;} CSend$

$$
\begin{array}{|l}
\multicolumn{1}{l}{S11} \\
\hline
\Delta FState; Args \\
\hline
sp? = rp? \land Op11 \Rightarrow st? \notin \operatorname{ran} ready' \land \\
\quad (\exists_1 p : PE \bullet c? \in \operatorname{dom}(scontrol'\, p)) \land (c?, st?) \in scontrol'\, sp? \land \\
\quad (\forall p : PE \bullet c? \notin \operatorname{dom}(rcontrol'\, p)) \land \\
\quad (sp?, c?) \in focus' \land \\
\quad rloc'\, sp?\, c? = rp? \land rloc'\, rp?\, c? = rp? \land \\
\quad (c?, st?) \in sender'\, sp? \land (c?, rt?) \in receiver'\, rp? \\
\hline
\end{array}
$$

Lemma 12 The $S12$ schema models a receive request at a local channel where no sender is waiting: the receiver is removed from the ready queue and blocked on the channel, the receiver location and identification are overwritten but not changed. The other functions carry over from *SStep*.

$Op12 \mathrel{\widehat=} SStep \mathbin{\S} CReceive$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;S12\;}} \\
\Delta FState; Args \\
\hline
Op12 \Rightarrow \\
rt? \notin \operatorname{ran} ready' \land \\
\quad (\forall p : PE \bullet c? \notin \operatorname{dom}(scontrol'\ p)) \land \\
\quad (\exists_1 p : PE \bullet c? \in \operatorname{dom}(rcontrol'\ p)) \land (c?, rt?) \in rcontrol'\ rp? \land \\
\quad c? \notin \operatorname{ran} focus' \land \\
\quad rloc'\ sp?\ c? = rp? \land rloc'\ rp?\ c? = rp? \land \\
\quad (c?, st?) \in sender'\ sp? \land (c?, rt?) \in receiver'\ rp? \\
\hline
\end{array}
$$

Lemma 13 As $S13$ shows, the implications of a send applied at a processor where the receiver is already waiting are the same as those in $S5$. The functions *scontrol*, *focus*, *rloc* and *receiver* keep their values from $S12$.

$Op13 \mathrel{\widehat=} S12 \mathbin{\S} CSend$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;S13\;}} \\
\Delta FState; Args \\
\hline
sp? = rp? \land Op13 \Rightarrow S5 \\
\hline
\end{array}
$$

Lemma 14 The schema $S14$ shows that when a receive request finds a sender already waiting then the implications are the same as those in $S3$.

$Op14 \mathrel{\widehat=} S11 \mathbin{\S} CReceive$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;S14\;}} \\
\Delta FState; Args \\
\hline
sp? = rp? \land Op14 \Rightarrow S3 \\
\hline
\end{array}
$$

Lemma 15 The $S15$ schema models a send request on a channel that has been used before and is known to have a remote receiver.

$Op15 \mathrel{\widehat=} SStep \mathbin{\S} CSend$

$$
\begin{array}{|l}
\multicolumn{1}{l}{S15} \\
\hline
\Delta FState;\ Args \\
\hline
sp? \neq rp? \land Op15 \Rightarrow \\
\quad st? \notin \operatorname{ran} ready' \land \\
\quad (\exists_1\, p : PE \bullet c? \in \operatorname{dom}(scontrol'\ p)) \land (c?, st?) \in scontrol'\ sp? \land \\
\quad (\forall\, p : PE \bullet c? \notin \operatorname{dom}(rcontrol'\ p)) \land \\
\quad c? \notin \operatorname{ran} focus' \land \\
\quad rloc'\ rp?\ c? = rp? \land \\
\quad (c?, st?) \in sender'\ sp? \land (c?, rt?) \in receiver'\ rp? \land \\
\quad focusm(rp?, c?, sp?) \in msgs! \\
\hline
\end{array}
$$

The sender is deleted from the ready queue and blocked on the channel, the sender identification is updated (but not changed) and a message is queued for transmission to convey the channel focus to the receiver's processor. The functions *rcontrol*, *focus*, *rloc* and *receiver* keep their values from *SStep*.

Lemma 16 The schema $S16$ shows what happens when a focus message, transmitted by a sender on a channel which has been used at least once before, arrives at the receiver's processor: the focus for the channel at $rp?$ is set and the sender location is set at $rp?$. The predicates on *ready*, *scontrol*, *rcontrol*, *sender* and *receiver* are all derived from $S15$.

$Op16 \mathrel{\widehat{=}} S15 \mathbin{;} CFocus$

$$
\begin{array}{|l}
\multicolumn{1}{l}{S16} \\
\hline
\Delta FState;\ Args \\
\hline
sp? \neq rp? \land Op16 \Rightarrow \\
\quad st? \notin \operatorname{ran} ready' \land \\
\quad (\exists_1\, p : PE \bullet c? \in \operatorname{dom}(scontrol'\ p) \land p = sp?) \land \\
\quad (\forall\, p : PE \bullet c? \notin \operatorname{dom}(rcontrol'\ p)) \land \\
\quad (\exists_1\, p : PE \bullet (p, c?) \in focus' \land p = rp?) \land sloc\ rp?\ c? = sp? \land \\
\quad (c?, st?) \in sender'\ sp? \land (c?, rt?) \in receiver'\ rp? \\
\hline
\end{array}
$$

Lemma 17 The $S17$ schema models a receive request being applied for a channel known to be remote and for which the focus has already arrived.

$Op17 \mathrel{\widehat{=}} S16 \mathbin{;} CReceive$

$$
\begin{array}{l}
\underline{\quad S17 \quad}\\
\Delta FState;\ Args\\
\hline
sp? \neq rp? \land Op17 \Rightarrow rt? \notin \operatorname{ran} ready' \land\\
\quad (\exists_1\, p : PE \bullet c? \in \operatorname{dom}(scontrol'\, p) \land p = sp?) \land\\
\quad (\exists_1\, p : PE \bullet c? \in \operatorname{dom}(rcontrol'\, p)) \land (c?, rt?) \in rcontrol'\, rp? \land\\
\quad c? \notin \operatorname{ran} focus' \land rloc'\, rp?\, c? = rp? \land\\
\quad (c?, st?) \in sender'\, sp? \land (c?, rt?) \in receiver'\, rp? \land\\
\quad rtr(sloc\, rp?\, c?, c?, rp?) \in msgs!
\end{array}
$$

The receiver thread is deleted from the ready queue and blocked on the channel, the focus is consumed, the receiver location and identification are updated (but not changed) and an *rtr* message is returned to the sender.

Lemma 18 The $S18$ schema gives the implications of a send and receive, in either order, on different processors: the sender and receiver are removed from their ready queues and blocked on the channel, the receiver location is overwritten (but not changed) at the receiver's processor, the sender and receiver identifications likewise, a message containing the channel focus is submitted at $sp?$ for transmission to $rp?$ but $focus'$ retains its value from the schemas $S12$ and $S15$.

$Op18 \mathrel{\widehat=} (S15 \mathbin{⨾} S12) \lor (S12 \mathbin{⨾} S15)$

$$
\begin{array}{l}
\underline{\quad S18 \quad}\\
\Delta FState;\ Args\\
\hline
sp? \neq rp? \land Op18 \Rightarrow st? \notin \operatorname{ran} ready' \land rt? \notin \operatorname{ran} ready' \land\\
\quad (\exists_1\, p : PE \bullet c? \in \operatorname{dom}(scontrol'\, p)) \land (c?, st?) \in scontrol'\, sp? \land\\
\quad (\exists_1\, p : PE \bullet c? \in \operatorname{dom}(rcontrol'\, p)) \land (c?, rt?) \in rcontrol'\, rp? \land\\
\quad c? \notin \operatorname{ran} focus' \land\\
\quad rloc'\, rp?\, c? = rp? \land\\
\quad (c?, st?) \in sender'\, sp? \land (c?, rt?) \in receiver'\, rp? \land\\
\quad focusm(rloc\, sp?\, c?, c?, sp?) \in msgs!
\end{array}
$$

Lemma 19 The schema $S19$ shows the consequences of processing a message containing the channel focus at the receiver's processor: an *rtr* message is submitted at the receiver's processor for transmission to the sender's processor. The values of the other functions are carried over from $S18$.

$Op19 \mathrel{\widehat=} S18 \mathbin{⨾} CFocus$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad S19 \quad}} \\
\Delta FState;\ Args \\
\hline
sp? \neq rp? \wedge Op19 \Rightarrow (\exists_1\, p : PE \bullet c? \in \mathrm{dom}(scontrol'\ p) \wedge p = sp?) \wedge \\
\quad (\exists_1\, p : PE \bullet c? \in \mathrm{dom}(rcontrol'\ p) \wedge p = rp?) \wedge \\
\quad rloc'\ rp?\ c? = rp? \wedge \\
\quad (c?, st?) \in sender'\ sp? \wedge (c?, rt?) \in receiver'\ rp? \wedge \\
\quad rtr(sp?, c?, rp?) \in msgs! \\
\hline
\end{array}
$$

Lemma 20 The $S20$ schema models the processing of an rtr message at a sender's processor: the sender is unblocked from the channel and returned to the ready queue, the receiver location is overwritten (but not changed) and the data message queued for transmission to the receiver's processor.

$Op20 \mathrel{\widehat{=}} (S17 \vee S19) \mathbin{⨾} CRtr$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad S20 \quad}} \\
\Delta FState;\ Args \\
st! : THREAD \\
\hline
sp? \neq rp? \wedge Op20 \Rightarrow S9 \\
\hline
\end{array}
$$

6.2.2 Communication proof

Having stated the assumptions about the sequences of communication operations that the Testbed may be required to execute, and having given a list of lemmas demonstrating the effects of applying various operations to various states, I now prove that the lemmas include all possible state transitions and that all transitions representing a communication involve synchronisation between sender and receiver.

The proof is by induction. The base case assumes that the channel has never been used before (as modelled by $SBase$) and the step case assumes that at least one communication has been completed on the channel (as modelled by $SStep$). Since channels operate independently from each other, the proof considers an arbitrary channel without loss of generality.

Communication proof base case: There are four sub-cases depending on whether the sender and receiver share a processor or not and depending on whether the sender requests to communicate first or second.

1. If the sender and receiver are executing on the same processor and the sender communicates first then Lemma 2 shows that the sender will be blocked. When the receiver requests communication, the situation is described by

Lemma 3 which shows that the sender is restarted. The suspension of the sender implements synchronisation.

2. If the sender and receiver share a processor but the receiver requests to communicate first, then Lemma 4 shows that the receiver will be blocked. When the sender requests communication, the situation is described by Lemma 5 which shows that the receiver is restarted. The suspension of the receiver implements synchronisation.

3. If the sender and receiver do not share a processor then it does not matter which requests to communicate first but it does matter whether the *advert* message issued by the sender is processed at the receiver's site before or after the receiver requests to communicate.

   Supposing the advertisement arrives before the receiver. Lemma 6 shows that in processing the advert message the location of the sender is stored so that in Lemma 7 the blocking receiver returns an *rtr* to the sender's processor. Lemmas 9 and 10 then show the processing of the *rtr* and the subsequent data message. Note that the sender is suspended until it receives the *rtr* message and that the receiver is suspended until it receives the data message—this implements synchronisation.

4. Finally, suppose that the advertisement arrives after the receiver requests to communicate. Lemma 4 shows the receiver being blocked and Lemma 8 the processing of the advert message. The argument now continues as above with the processing of the *rtr* message.

Communication proof step case: I assume that all communications in the step case begin in the state characterised by *SStep* and I prove that they all end in state *SStep*. Note that all communications in the base case end in the *SStep* state (the post conditions of $S3$, $S5$ and $S10$ all contain the conditions of *SStep*). There are four sub-cases much as before.

1. If the sender and receiver are executing on the same processor and the sender communicates first then Lemma 11 shows that the sender will be blocked. When the receiver requests communication, the situation is described by Lemma 14 which shows that the sender is restarted. The suspension of the sender implements synchronisation. The post conditions of $S14$ are a superset of *SStep*.

2. If the sender and receiver share a processor but the receiver requests to communicate first, then Lemma 12 shows that the receiver will be blocked. When the sender requests to communicate, the situation is described by Lemma 13 which shows that the receiver is restarted. The suspension of the receiver implements synchronisation. The postconditions of $S13$ are a superset of *SStep*.

3. If the sender and receiver do not share a processor then it does not matter which requests to communicate first but it does matter whether the focus message issued by the sender is processed at the receiver's site before or after the receiver requests to communicate.

   Supposing the focus arrives before the receiver. Lemma 16 shows that in processing the focus message the location of the sender is stored so that in Lemma 17 the blocking receiver returns an *rtr* to the sender's processor. Lemmas 20 and 10 then show the processing of the *rtr* and the subsequent data message. Note that the sender is suspended until it receives the *rtr* message and that the receiver is suspended until it receives the data message—this implements synchronisation. The postconditions of $S10$ are a superset of $SStep$.

4. Finally, suppose that the focus arrives after the receiver requests to communicate. Lemma 12 shows the receiver being blocked and Lemma 19 the processing of the focus message. The argument now continues as above with the processing of the *rtr* message.

This completes the proof. □

6.3 Transparency of Thread Migration

This subsection follows a similar format to the preceding one. A series of lemmas show, for different beginning states, what happens to channel communication when a thread migrates. The proof then demonstrates that all possible 'before' states have been considered and that in each case migration is transparent, i.e. the 'after' state of the operation is essentially the same as the before state.

The effects of migration are considered in relation to a single, arbitrarily chosen channel so the before states in the proof base case are a subset of those presented in Lemmas 2 to 20. In fact, the before states are a strict subset because migration can only occur if the migrating thread is in the ready queue (and not blocked on the channel). The before states are subject to schema renaming—the migration schemas presented below use the three input arguments $mt?$, $fp?$ and $tp?$ to represent the migrating thread, the 'from processor' and the 'to processor' respectively—so if the migrating thread is a sender for $c?$ then $mt?$ is identified with $st?$ and $fp?$ is identified with $sp?$, otherwise if the migrating thread is a receiver for $c?$ then $mt?$ is identified with $rt?$ and $fp?$ is identified with $rp?$.

6.3.1 The receiver location problem

The after states are subject to the same rules for renaming and, as will be shown, are identical in all relevant aspects to the corresponding before states. In most cases it is obvious that the functions in the after state have the same value as they did in the before state. The receiver location *rloc*, however, is updated in a more complicated fashion and therefore requires some further explanation.

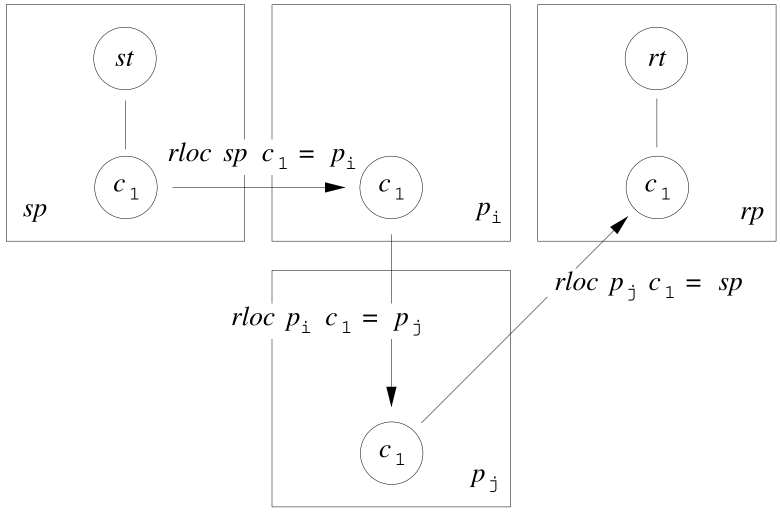

Figure 10: *A receiver rt executing on processor p_i communicates with sender st before migrating to p_j and then rp. Forwarding addresses are left at p_i and p_j so the sender can still locate the receiver when necessary.*

For the first communication over a channel, senders locate the appropriate receiver by broadcasting their presence to all processors in the system. For reasons of efficiency, second and subsequent communications rely on the sender being able to transmit the channel focus more directly to the receiver. When migration is disallowed, the sender simply remembers the last location of the receiver and sends the focus there—this is how *rloc* has been used in the lemmas up to this point.

However, when migration is allowed, a receiver may migrate from processor to processor between two communications and thus cause the sender to transmit its focus to the wrong place. The solution employed on the Testbed is to have receivers leave a forwarding address behind them every time that they migrate. Now, when a sender transmits a focus to an old address it can be forwarded as necessary until it catches up with the receiver. The receiver location *rloc* is used to encode these forwarding addresses as shown in Figure 10.

To make it easier to compare *rloc* functions in the before and after states I define a new predicate called linked. A focus message from a sender on processor $sp?$ for a channel $c?$ will reach a receiver on $rp?$ if the receiver's processor is linked to the sender's processor, i.e. if $rp? \in \mathsf{linked}(rloc, sp?, c?)$.

$$\mathsf{linked} : ((PE \nrightarrow (CHANNEL \nrightarrow PE)) \times PE \times CHANNEL) \nrightarrow \mathbb{P}\, PE$$

$$\forall\, r : PE \nrightarrow (CHANNEL \nrightarrow PE);\ p : PE;\ c : CHANNEL \bullet$$
$$\quad \mathsf{linked}(r, p, c) = \mathbf{if}\ c \in \mathrm{dom}(r\ p)\ \mathbf{then}\ \{r\ p\ c\} \cup \mathsf{linked}(r, r\ p\ c, c)\ \mathbf{else}\ \varnothing$$

6.3.2 Lemmas

The next group of lemmas follow the format already established: a new schema Opn is defined in terms of $MConnect$ and $MDisconnect$ applied to a previously defined state and a state schema Sn is defined in which the relevant consequences of Opn are shown.

Lemma 21 Neither migrate schema in $Op21$ changes $scontrol$ or $rcontrol$, so they maintain their values from $SStep$.

$Op21 \mathrel{\widehat=} SStep[mt?/st?, fp?/sp?] \mathbin{\text{⨟}} MDisconnect \mathbin{\text{⨟}} MConnect$

$$
\begin{array}{|l}
\multicolumn{1}{l}{S21} \\
\hline
\Delta FState; Args \\
mt? : THREAD \\
fp?, tp? : PE \\
\hline
Op21 \Rightarrow \\
\quad (\forall p : PE \bullet c? \notin \mathrm{dom}(scontrol'\, p) \land c? \notin \mathrm{dom}(rcontrol'\, p)) \land \\
\quad c? \notin \mathrm{ran}\, focus' \land (rp?, c?) \in used' \land \\
\quad rloc'\, tp?\, c? = rp? \land rloc'\, rp?\, c? = rp? \land \\
\quad (c?, st?) \in sender'\, tp? \land (c?, rt?) \in receiver'\, rp? \\
\hline
\end{array}
$$

The channel $c?$ is in the sets named sc and scl but not in rc or rcl so the functions $focus$ and $used$ are not updated with regard to $c?$. Function $rloc$ is not updated at $fp?$ but is overwritten with $tp? \mapsto \{c \mapsto p\}\}$ by $MDisconnect$ so if $tp? = rp?$ then $rloc\, rp?\, c? = rloc\, fp?\, c? = rp?$ otherwise $rloc\, rp?\, c?$ maintains its previous value. The function $sender$ is updated at $tp?$ in the correct way, $receiver$ retains its original value.

Lemma 22 The migrate operation $Op22$ does not update $rcontrol$ or $scontrol$ so their values are maintained from $S12$. The channel $c?$ is in the sets sc and scl but not in rc or rcl so the function $focus$ is not updated with regard to $c?$. $MConnect$ ensures that the sender's new processor knows $rloc\, tp?\, c? = rp?$ and, if $rp? = tp?$ this ensures $rloc\, rp?\, c? = rp?$ as well—otherwise, $rloc\, rp?$ retains its value from $S12$. The function $sender$ is updated at $tp?$ in the correct way, $receiver$ retains its original value.

$Op22 \mathrel{\widehat=} S12[mt?/st?, fp?/sp?] \mathbin{\text{⨟}} MDisconnect \mathbin{\text{⨟}} MConnect$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad S22 \quad}} \\
\Delta FState; Args \\
mt? : THREAD \\
fp?, tp? : PE \\
\hline
Op22 \Rightarrow (\forall\, p : PE \bullet c? \notin \mathrm{dom}(scontrol'\, p)) \land \\
\quad (\exists_1\, p : PE \bullet c? \in \mathrm{dom}(rcontrol'\, p)) \land (c?, rt?) \in rcontrol'\, rp? \land \\
\quad c? \notin \mathrm{ran}\, focus' \land \\
\quad rloc'\, tp?\, c? = rp? \land rloc'\, rp?\, c? = rp? \land \\
\quad (c?, st?) \in sender'\, tp? \land (c?, rt?) \in receiver'\, rp? \\
\hline
\end{array}
$$

Lemma 23 The migrate operation $Op26$ does not update *rcontrol* or *scontrol* so their values are maintained from $S11$. The channel $c?$ is in the sets *rc* and *rcl* but not in *sc* or *scl* so the focus is removed at processor $fp?$ and created again at $tp?$. The receiver location at $tp?$ is updated by *MConnect* to be $tp?$. The function *receiver* is updated at $tp?$ in the correct way, *sender* retains its original value.

$Op23 \mathrel{\widehat=} S11[mt?/rt?, fp?/rp?] \mathbin{⨟} MDisconnect \mathbin{⨟} MConnect$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad S23 \quad}} \\
\Delta FState; Args \\
mt? : THREAD \\
fp?, tp? : PE \\
\hline
Op23 \Rightarrow (\exists_1\, p : PE \bullet c? \in \mathrm{dom}(scontrol'\, p)) \land (c?, st?) \in scontrol'\, sp? \land \\
\quad (\forall\, p : PE \bullet c? \notin \mathrm{dom}(rcontrol'\, p)) \land \\
\quad (tp?, c?) \in focus' \land \\
\quad (c?, st?) \in sender'\, sp? \land (c?, rt?) \in receiver'\, tp? \\
\hline
\end{array}
$$

Lemma 24 The migrate operation $Op24$ does not update *scontrol* or *rcontrol* so their values are maintained from $S15$.

$Op24 \mathrel{\widehat=} S15[mt?/rt?, fp?/rp?] \mathbin{⨟} MDisconnect \mathbin{⨟} MConnect$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad S24 \quad}} \\
\Delta FState; Args \\
mt? : THREAD \\
fp?, tp? : PE \\
\hline
Op24 \Rightarrow (\exists_1\, p : PE \bullet c? \in \mathrm{dom}(scontrol'\, p)) \land (c?, st?) \in scontrol'\, sp? \land \\
\quad (\forall\, p : PE \bullet c? \notin \mathrm{dom}(rcontrol'\, p)) \land \\
\quad rloc'\, tp?\, c? = tp? \land \\
\quad (c?, st?) \in sender'\, sp? \land (c?, rt?) \in receiver'\, tp? \land \\
\quad (\exists\, p : PE \bullet focusm(p, c?, sp?) \in msgs! \land rp? \in \mathsf{linked}(rloc, p, c?)) \\
\hline
\end{array}
$$

The channel $c?$ is in the sets rc and rcl but not in sc or scl so the receiver location recorded at $tp?$ is set to $tp?$ by $MConnect$. As before, $receiver$ is updated by the appropriate value at $tp?$ and $sender$ is left unchanged. The focus message is still waiting to be delivered, but to the processor that the receiver has just left. However, the $MDisconnect$ operation sets $rloc\, fp?\, c?$ to $tp?$ which, when $CFocus$ is applied at $fp?$, will cause the first implication to be true and the focus message to be forwarded to $tp?$.

Lemma 25 The migrate operation $Op25$ does not update $scontrol$ or $rcontrol$ so their values are maintained from $S16$. The channel $c?$ is in the sets rc and rcl but not in sc or scl so the focus is deleted at $fp?$ and restored at $tp?$. As before, $receiver$ is updated by the appropriate value at $tp?$ and $sender$ is left unchanged.

$Op25 \mathrel{\widehat=} S16[mt?/rt?, fp?/rp?] \fatsemi MDisconnect \fatsemi MConnect$

$$
\begin{array}{|l}
\underline{\;S25\;}\\
\Delta FState;\ Args\\
mt? : THREAD\\
fp?, tp? : PE\\
\hline
Op25 \Rightarrow (\exists_1\, p : PE \bullet c? \in \mathrm{dom}(scontrol'\, p) \land p = sp?) \land\\
\qquad (\forall\, p : PE \bullet c? \notin \mathrm{dom}(rcontrol'\, p)) \land\\
\qquad (tp?, c?) \in focus' \land\\
\qquad (c?, st?) \in sender'\, sp? \land (c?, rt?) \in receiver'\, tp?
\end{array}
$$

6.3.3 Migration proof

I now show the transparency of migration with respect to communication by demonstrating that the lemmas presented above exhaust all the situations in which migration may occur and by showing that the postconditions of the lemma state schemas are essentially the same as the before state of the lemma operations. The proof is by induction: the base case models the first migration after a communication on $c?$ and the step case models the second or subsequent migration after the last communication on $c?$.

It is assumed that no operations occur on the channel in question between the application of $MDisconnect$ and the application of $MConnect$—although this is not true for the Testbed the proof can be extended to cover this case relatively easily.

Migration base case: There are a number of sub-cases to consider depending on whether $mt?$ is a sender or a receiver for the channel and whether the other user of the channel has requested to communicate yet or not.

1. If $mt?$ is the sender for $c?$ then there are two situations depending on whether the receiver has requested to communicate yet.

   If the receiver has not yet requested to communicate then the state of $c?$ is described by $SStep$ and the effect of the migration operation by $Op21$ in Lemma 21. It does not matter where the receiver is located. The postconditions of $S21$ contain the conditions of $SStep[mt?/st?, tp?/sp?]$ so in this case migration is transparent.

2. If $mt?$ is the sender for $c?$ and the receiver has already requested to communicate then the state of $c?$ is described by $S12$ and the effect of the migration operation $Op22$ by Lemma 22. It does not matter where the receiver is located. The postconditions of $S22$ contain the conditions of $S12$ so the migration is transparent.

3. If $mt?$ is the receiver for $c?$ then there are three situations depending on whether the sender is local or not and for remote senders whether the focus message has arrived or not.

   If the sender is local and has already requested to communicate then the state of $c?$ is described by $S11$ and the effect of the migration operation by $Op23$ in Lemma 23. The postconditions of $S23$ do not contain the conditions of $S11$ but they do contain the conditions of $S16$ and $S16$ represents essentially the same point in the communication protocol, so the migration is transparent. (In $S16$ there is a focus at the receiver's processor so the receiver can deduce that there is a waiting sender and the sender location points to the sender's processor so that an rtr message can be sent to the correct location.)

4. If $mt?$ is the receiver for $c?$ and the sender is not local, has already requested to communicate but its focus message has not arrived, then the state of $c?$ is described by $S15$ and the effect of the migration operation by $Op24$ in Lemma 24. The postconditions of $S24$ contain the conditions of $S15$ modified so that the focus message is being sent indirectly to the receiver.

5. If $mt?$ is the receiver for $c?$ and the sender is not local, has already requested to communicate and its focus message has arrived, then the state of $c?$ is described by $S16$ and the effect of the migration operation by $Op25$ in Lemma 25. The postconditions of $S25$ contain the conditions of $S16$ so migration is transparent.

Migration step case: The step case states that the above arguments also hold for the second or any subsequent migration. This completes the proof. $\square$

7 Conclusions

A formal specification in **Z** was developed in Sections 4 and 5, focusing on those parts of the Testbed operating system concerned with task exchange, i.e. thread synchronisation, channel communication and thread migration. Framing schemas were developed to show how sequences of operations are built up. Section 6 then used the specification to demonstrate properties of safety and correctness—an assurance that, provided certain constraints are met, user tasks can be migrated as many times as is desired without changing the results of their computation. A critical assessment of the **Z** language is presented next, after the advantages and disadvantages of formal specification in general are discussed.

The formal specification was carried out in tandem with the development of the Testbed's operating system. This was beneficial in several ways. Firstly, the specification provided a much faster way of prototyping procedures for the implementation than the traditional code, test and debug cycle. Secondly, the specification coped well with changes in the requirements for the implementation and was invaluable for indicating, albeit in a non-automatic way, ambiguities and inconsistencies in those requirements. Thirdly, the specification was useful as a tool for documentation, producing a clear description of the protocols being implemented. This meant not only that walk-throughs of the code were simplified but that the answers to various 'what if' questions could be given without having to write and execute test programs. As described in Martin [10] the specification also provided a precise way of presenting and interpreting performance measurements made of the intrinsic properties of the Testbed. Finally, the proofs showed the implementation to be safe and correct (provided the implementation is a correct reification of the specification) to a degree that could never have been determined by enumerating interactions between the multiple processors.

A single caveat in the use of formal specification: it is not always easy to find the most appropriate level of abstraction in preliminary versions of a specification. The work in Sections 4 and 5 went through a number of revisions before an acceptable balance was found between specifying the design principles and specifying features of the implementation. The verification process was helpful in indicating beneficial modifications to the specification.

I am still convinced that the **Z** language is one of the best for specifying task migration protocols. However, there are a number of areas in which **Z** needs to be used carefully. **Z** is a rich language, it has most of the operations a specifier needs already defined. Arbitrary use of different operations can be confusing for the reader of a specification and the work in this report has shown that it is very beneficial to impose a 'house style' covering choice of operators, order of presentation and naming of user-defined objects. The value of having a house style is also emphasised in Macdonald [9] where a lengthy list of good practices is given.

The **Z** language has associated tools for syntax and type checking and for typesetting. These allow a high degree of consistency and evenness of appearance in the final specification which aid the reader in navigating the document. Prototyping and refinement were not attempted in this report, so the lack of tools to assist these processes was irrelevant. The lack of a theorem prover for **Z** would only become a problem if the rigorous proofs presented in Section 6 were to be made formal proofs.

In conclusion, formal methods provide an invaluable way of proving safety and correctness of complicated protocols, such as those for thread migration. Formal methods also provide a host of other benefits: faster implementation, better documentation and a precise basis for presenting and interpreting performance measurements.

Acknowledgements This work was funded by a Research Studentship from the Science and Engineering Research Council.

References

[1] Antoni Diller. *Z: An Introduction to Formal Methods.* John Wiley & Sons, 1990.

[2] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7:11–19, September 1990.

[3] Ian Hayes. VDM and Z: A Comparative Case Study. *Formal Aspects of Computing*, 4(1):76–99, 1992.

[4] Ian J. Hayes. Applying Formal Specification to Software Development in Industry. *IEEE Transactions on Software Engineering*, 11(2):169–178, February 1985.

[5] Roland N. Ibbett, D. A. Edwards, T. P. Hopkins, C.K Cadogan, and D. A. Train. Centrenet—A High Performance Local Area Network. *The Computer Journal*, 28(3):231–242, July 1985.

[6] Kayhan Imre. *A Performance Monitoring and Analysis Environment for Distributed Memory MIMD Programs.* PhD thesis, University of Edinburgh, 1993.

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[8] INMOS Limited. *occam2 Reference Manual.* Prentice Hall International (UK) Ltd, 1988.

[9] R. Macdonald. Z Usage and Abusage. Technical Report Report 91003, Royal Signals and Radar Establishment, St Andrews Road, Malvern, Worcs WR14 3PS, February 1991.

[10] Paul Martin. The Performance Profiling of a Load Balancing Multicomputer. Technical Report ECS-CSG-3-94, University of Edinburgh, June 1994.

[11] Bertrand Meyer. On Formalism and Specifications. *IEEE Software*, 2(1):6–26, January 1985.

[12] David Pitt and Paddy Byers. The Rest Stays Unchanged. *(to appear in) Formal Aspects of Computing*, 1994.

[13] Dick Pountain. *A Tutorial Introduction to Occam Programming.* INMOS Limited, March 1987.

[14] J. M. Spivey. *The fUZZ Manual.* J. M. Spivey Computing Science Consultancy, Oxford, 1988.

[15] J. M. Spivey. *The Z Notation.* Prentice Hall, 1992.

[16] Peter H. Welch. The Role and Future of Occam. Available by ftp from unix.hensa.ac.uk (129.12.21.7), 1993.

[17] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer Magazine*, 23(9):8–24, 1990.

[18] Simon Woods. Error Diagnosis in a Distributed Environment. Master's thesis, University of Edinburgh, September 1990.

A Index of Z Terms