

Framework Applications for Parallel Computers

by
Schrettner, Lajos

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Department of Computer Science
József Attila University
August, 1999

*I would like to thank Dr. Innes Ritchie
for supporting me over the years. Her help,
encouragement and patience have been invaluable
to me since I started working on these projects.*

Contents

1	Introduction	1
1.1	Parallel hardware	1
1.2	Parallel software, programming models	2
1.2.1	Extensions of the von Neumann model	3
1.2.2	Declarative models	4
1.3	Decomposition and load balancing	4
1.4	Performance of parallel programs	5
1.5	Outline of the dissertation	5
2	Database query language executor	7
2.1	Introduction	7
2.2	OFL specification	8
2.2.1	Objects	8
2.2.2	Object references	8
2.2.3	Functions	8
2.2.4	OFL grammar	10
2.2.5	Database management with OFL	10
2.3	Parallel executor for OFL programs	14
2.3.1	Parallel Database Machine	14
2.3.2	Functional unit structure	14
2.3.3	OFL abstract machine	15
2.3.4	Preprocessing of OFL programs	17
2.4	Implementation	19
2.4.1	OFL Translator	19
2.4.2	OFL executor	22
2.4.3	Tests	25
3	Dynamic load balancing system	27
3.1	Introduction	27
3.2	Test environment	28
3.2.1	Decomposable applications	29
3.2.2	Test environment components	31
3.3	Processor commune	32
3.3.1	Load balancing strategies	32
3.3.2	Commune load balancing strategy	34
3.4	Tests	37
3.4.1	Test data	37
3.4.2	Processor topology	38

3.4.3	Results	38
3.5	Application evaluators	40
3.5.1	Application mimic	40
3.5.2	PROLOG interpreter	41
3.5.3	Attribute grammar evaluation	41
3.6	Implementation	42
4	Correctness of the load balancing protocol	43
4.1	Introduction	43
4.2	Problem description	44
4.3	Dynamic interaction skeleton	45
4.3.1	High-level structure	45
4.3.2	Structure of the component — the firm	45
4.3.3	The skeleton	46
4.4	Basic dynamic interaction — BDI	48
4.5	Bounded-resource dynamic interaction — BRDI	49
4.6	General dynamic interaction — GDI	51
4.7	A building block for larger systems	53
4.8	Applying the building block	54
5	Implementation of the load balancing system	55
5.1	Configuration	55
5.2	Initialisation	57
5.3	Profiling	58
5.3.1	Clock skew	59
5.3.2	Computing clock differences	59
5.4	Startup	59
5.4.1	Process hierarchy	60
5.4.2	Distribution/collection method	61
5.5	Task store	62
5.6	Application	63
5.7	Feeder	63
5.8	Load balancing components	64
6	Node processing model	65
6.1	Introduction	65
6.2	The NOP model	66
6.2.1	Data	66
6.2.2	Parallelism	67
6.2.3	Transformations	68
6.2.4	Discussion	69
6.3	Programs	70
6.3.1	Example	71
6.3.2	High level NOP language	72
6.3.3	Parameterised transformations	72
6.3.4	Transformation name parameters	74
6.4	High level programming	76
6.4.1	Enumeration	76
6.4.2	Producer-consumer pairs	77
6.4.3	Divide and conquer	78

6.4.4	NOP quick sort	78
6.5	Implementation	80
6.5.1	Data representation	80
6.5.2	Execution units	80
6.5.3	Cost of transformations	81
6.5.4	Multiprocessor implementation	81
6.5.5	Code listing guide	82
7	Summary	83
8	Összefoglaló (Summary in Hungarian)	87
	References	91
	Appendices	
A	OFL executor code listing	1
A.1	Parser	1
A.2	OFL Abstract Machine Code Generator	4
A.3	Executor structure	9
A.4	OFL Abstract Machine	13
A.5	Database Access	17
A.6	Translated OFL program	19
B	Load balancing test system code listing	21
B.1	Configuration	22
B.2	Host connection	23
B.3	Startup	26
B.4	Profiler	28
B.5	Store	30
B.6	Application	34
B.7	Boss	35
B.8	Sentinel	37
C	NOP executor code listing	41
C.1	Data representation	42
C.2	Transformation representation	43
C.3	Memory management	45
C.4	Translation	47
C.5	Executor	56

Chapter 1

Introduction

Parallel computers and parallel programming have received considerable attention in recent years. This can be attributed to two factors. First, the ever growing need for processing power makes it necessary to look for new architectures because serial computers are approaching physical limits. It seems that the only way to increase the performance of computers is by using multiple processing elements which operate concurrently, in parallel. Second, the history of computing taught us that hardware and software are equally important, the complexity of problems can only be tackled by applying appropriate methods to design our systems. Parallelism may help allowing us to model real world and artificial phenomena in a natural way, thereby leading to cleaner, more reliable applications.

Besides its advantages, parallelism has drawbacks. The coordination of concurrent activities poses new problems and requires new methods, but these are inherently more difficult than in the sequential case. Further, despite its potential, parallel programming has not become widespread because there are no universally accepted general purpose parallel programming models. A large number of architectures and programming languages have been proposed, but at present, applications developed for a given architecture can only be **ported to other architectures with substantial effort**.

In the dissertation, I study several aspects of the parallel computing field, present applications that are the results of my work in this area and propose a parallel programming model (NOP) which can serve as a general programming model. The applications are not designed to solve a single problem, but provide a framework encapsulating functions that can be used to implement a range of systems similar in their characteristics.

1.1 Parallel hardware

Parallel computer architectures can be categorised in several ways depending on what characteristics are considered [14]. A parallel architecture consists of processors, memory module(s) and optionally some kind of interconnection mechanism to connect processors and/or processors and memory modules [10]. The simplest and most popular classification [11] distinguishes four classes based on the number of independent instruction and data streams: SISD (Single Instruction/Single Data stream), SIMD (Single Instruction/Multiple Data stream), MISD (Multiple Instruction/Single Data stream), and MIMD (Multiple Instruction/Multiple Data stream). SISD machines are the traditional sequential von Neumann machines. The SIMD category [13] contains vector computers, where a single instruction is executed on multiple data items in a synchronised manner. There is no con-

sensus on what the MIMD class covers, most authors do not put any existing machines into it. MIMD machines contain autonomous processing elements (PEs, processors with private memory) working asynchronously.

Multiple data stream machines can be further classified based on where the data streams originate from. Shared memory (SM) machines have a memory module that all processors use for data storage. Multiple accesses to the same memory location have to be coordinated in order to avoid data corruption. In distributed memory (DM) machines each processor has only private memory. Nevertheless, PEs have to cooperate to achieve a common goal, so these machines contain an interconnection facility which enables PEs to exchange data. PEs can be connected by point-to-point communication links or they all can be connected to a broadcast network. The most widely used parallel machines are SM-MIMD (eg. multiuser UNIX machines, even multiprocessor PCs) and DM-MIMD, although we can find examples for others as well [13].

Ideally, parallel machines should be extensible, ie. users should be able to add PEs and/or memory to increase processing power. The SM-SIMD architecture has two disadvantages, both limiting its extendibility [10][12]. First there is a need for global synchronisation among processors, this becomes more and more problematic as the number of processors grows. Second, the shared interface to the global memory forms a bottleneck. To some extent, the bottleneck can be handled by using multistage switches, multiple memory banks and caching. Sadly, cache coherence is hard to maintain due to random accesses to memory. Cache performance is better if successive accesses are localised both in terms of processors and memory locations. It seems that distributed memory is more favourable in this respect, but programming with shared memory is easier because of its similarity to conventional sequential computer programming.

The applications to be described have been developed on a transputer network [15][16][17][18][20]. A transputer network is a perfect example of a DM-MIMD machine, each processor has private memory, PEs are connected via point-to-point links. In contrast to SM-MIMD machines, DM-MIMD machines are easy to extend using only a constant number of communication links per processor.

1.2 Parallel software, programming models

The main obstacle in the way of widespread use of parallel machines is that software development is fundamentally more difficult. Owing to the more complex structure of multicomputers, new language constructs, new software development methods have to be devised. The situation is similar to the one a few decades ago, when the software crisis stimulated the birth of software engineering. An active area of research in parallel software engineering [22][23][26] tries to explore the methods and tools that can be used to engineer reliable parallel systems. By now there is a great deal of knowledge and experience related to sequential algorithms, but parallel machines and programming are relatively new. There are many competing architectures, programming concepts and languages, but there are no universally accepted, general parallel programming models [27][28].

It is important that we make a distinction between architectures and programming models. A programming model is an abstract machine with its data structures and operations that connects the physical machine and the higher level language(s) used by humans. It has to meet conflicting requirements from the two sides, it has to be convenient to use for problem solving and has to allow efficient implementation(s). In sequential programming, the imperative programming model (the von Neumann model) is successful because high-level imperative languages can efficiently be compiled into machine code. In

parallel programming, those models that allow us to write efficient programs use low-level primitives, while high-level paradigms are not universally applicable.

Existing parallel models can be categorised in various ways, but two categories seem to be the most important to discuss. The first category contains extensions to the von Neumann model. They inherit a number of features from it: they are imperative and use random access to fetch and store data from/to memory. The imperative style allows very efficient programs to be written, sometimes with considerable effort. Programmers are used to this style, although another important category of models, the declarative programming models [29][30][31][32] are gaining on. They have a number of advantages over the von Neumann model even if we are interested only in sequential implementations. The functional model for example provides explicit representation for data structures, allows functions to be treated as values and exhibits referential transparency. Problems (not directly related to parallel programming) that can cause inefficiencies are the lack of selective update of data structures and the missing concept of computational state.

1.2.1 Extensions of the von Neumann model

PRAM: Parallel Random Access Machine [34]. The data structure is global random access memory, it has a number of processors working in a synchronised manner. It is easy to program, but harder to implement. The most suitable architecture for it is the SM-SIMD, in fact there is an obvious one-to-one correspondence between the components.

Process models

Processes are independent, asynchronous threads of control. Process models usually augment the von Neumann model with operations handling process creation/destruction and synchronisation/communication. A serious drawback common to all process models is that the execution of programs is very sensitive to timing conditions, making the design and testing of programs very hard. Asynchronous processes are very useful in certain application areas (eg. real-time systems), but they seem to cause more problems than they solve in others.

Shared memory process models: Various models use various methods for synchronisation, eg. semaphores, monitors [35] and derivatives of these. Shared memory models are very close to the SM-MIMD architecture, with all the problems of shared memory (see above). Attempts to simulate shared memory over physical distributed memory have not provided an efficient general solution either.

Distributed memory process models [36]: Here processes communicate dominantly by sending messages to each other. The models are very general and powerful: extensions of existing languages provide models in this category: well known are the PVM [37] and MPI libraries; more formal ones are Distributed Processes [38] and CSP [39]. These models are well suited to SM-MIMD and DM-MIMD architectures. OCCAM [45][46][47] is a parallel programming language based on CSP. The transputer and OCCAM were designed together to provide a uniform platform for parallel applications. INMOS Parallel C [48] is an other language for the transputer, it is a standard ANSI C implementation with added libraries for process control and communication facilities.

Other high-level models

While the models above were probably influenced by their underlying architectures, the ones in this group are farther away from the hardware.

BSP: The Bulk-Synchronous Parallel [27] model consists of PEs, memory units, a router, and a synchronising facility. Computation consists of supersteps with synchronisation at regular intervals to determine the end of each superstep. With careful adjustment of machine parameters, good implementations are possible on SM-MIMD and DM-MIMD architectures. BSP is a very promising approach, the construct to express parallelism in the NOP model resembles BSP supersteps.

Linda: Linda has a global Tuple Space (TS) as its data structure, processes insert, read and remove tuples to/from TS. It is conceptually simple from the programmers point of view and allows several implementation strategies [41].

1.2.2 Declarative models

In these models [29][30][31][32] the data structures are certain aggregates of elementary types, the operations can access and operate on data that are passed to them, thus memory references are localised and exclusive (cf. caching in SM machines). Further, as their name suggests, they do not completely specify the sequence of events during program execution (evaluation). These two factors (localised references and incomplete sequencing) allow independent subproblems to be detected and evaluated in parallel [42][43][44]. These models are best suited for SM-MIMD machines, but MD-MIMD implementations are possible using dynamic load balancing techniques. There are attempts to design special purpose architectures as well.

1.3 Decomposition and load balancing

The central idea of parallel computation is that multiple processors work simultaneously to achieve a final result. For this to be possible, the problem to be solved has to be decomposed into independent parts: functional and/or domain decomposition can be used to achieve this.

In the case of *functional decomposition* the algorithm is partitioned and possibly assigned to different processors. Data are routed from one processor to the other as necessary during the computation. One popular form of functional decomposition is the *pipeline*, where processors are arranged in an assembly line fashion, each performing one particular transformation on data flowing through it. The database query language executor described in the dissertation uses a pipeline arrangement (Chapter 2). *Domain decomposition* means that data is partitioned into subsets which then can be processed more or less independently. When using data decomposition, often all processors execute the same program, this is called the SPMD (Single Program/Multiple Data) paradigm. This approach was used in the load balancing environment, the other application presented in the dissertation (Chapter 3).

Load balancing is the activity of ensuring that all processors perform useful work during the execution time of a given program. Some problems have a regular structure that allows us to define a decomposition in advance, before the program runs. In these cases load balancing is part of the design phase and is called *static load balancing*. Image processing problems, for example, tend to belong to this class as very often a large image can be

decomposed into rectangular areas and processed in parallel. On the other hand, there are several important problems that cannot be decomposed in advance, therefore load balancing becomes an integral part of the algorithm. This is called *dynamic load balancing*. The performance of such an algorithm mainly depends on the effectiveness of load balancing. A widely used dynamic load balancing method is the so-called processor farm [49]. The particular load balancing method described in the dissertation can be regarded as a generalisation of the farm principle.

1.4 Performance of parallel programs

The primary aim of parallelisation is faster program execution, so we are most interested in the *running time* of parallel programs. Running time is influenced not only by the input data, but by the number of processors used. Other measures also yield useful information about the behaviour of programs [50]. *Speedup* is defined as the ratio of the running time of the fastest sequential program and the running time of the parallel program. This ratio indicates the improvement in solution time using parallelism. *Efficiency* is defined as the ratio of the speedup and the number of processors. It is an important measure of performance because it shows how effectively the processors are used. Higher efficiency means better utilisation, in the best case its value is 1.

1.5 Outline of the dissertation

Chapter 2 summarizes work carried out under the Copernicus project *Large Parallel Databases*. The most important results were

- ▶ specification and formal definition of the Object Functional Language (OFL)
- ▶ proposal for an extended OFL-based **database machine**
- ▶ design of an abstract machine to serve as a basis for the definition and implementation of the language
- ▶ multiprocessor implementation of an OFL executor
- ▶ tests showed that the executor was able to reduce execution time

Chapter 3 deals with dynamic load balancing, in particular it

- ▶ defines the class of (special) decomposable problems
- ▶ describes a test environment implemented to enable the investigation of special decomposable problems
- ▶ introduces the *processor commune* model as a generalisation of the processor farm model
- ▶ reports test results which indicate that the processor commune model is effective for the execution of declarative languages

Chapter 4

- ▶ defines a building block that can be used to construct deadlock-free processes
- ▶ shows by successive refinement and case analysis that the construction is correct
- ▶ shows how the processor commune can be built from the building block

Chapter 5 explains the structure of the processor commune implementation on a transputer network highlighting some of the difficulties and design decisions.

Chapter 6 introduces the NOP (NDe Processing) model

- ▶ defines the data structures and transformations
- ▶ shows how it relates to existing models
- ▶ describes a high level language based on the model
- ▶ gives examples of how problems can be solved with the model
- ▶ shows how the model can be implemented on shared memory multiprocessors

Programs are listed in the Appendices.

Chapter 2

Database query language executor

Based on [1][2].

2.1 Introduction

In 1994–95 a team of department members took part in the Copernicus project named Large Parallel Databases (LPD). The aim of the collaborating parties was to explore diverse aspects of parallel database technology and provide both hardware and software solutions to identified problems. Our team ventured on creating a suite of tools for a database query language drafted by another member of the project.

Object oriented database management systems are based on an underlying object oriented data model and play an important role in the efficient management of structured objects [51]. The object oriented data model supports the construction of complex objects through user defined types. Similar objects are grouped together in classes encapsulating methods and attributes. Users of such databases express their queries in OQL (Object Query Language). The LPD project members decided that it was advantageous to translate OQL queries into an intermediate form, into expressions given in Object Functional Language (OFL).

The first task was to define the OFL language precisely based on earlier drafts [52][53]. As a result, a specification and formal definition for the syntax and semantics of the OFL language was given. It identified the data objects, object references and functions comprising the language. It clearly separated the components that belong to the language and those of the environment in which programs are run. The grammar was given in a form acceptable to the PROF-LP compiler generator system [54].

A low-level abstract machine was designed to serve as a basis for the definition of semantics and to ease the implementation of OFL. It was a convenient target for compilation and allowed efficient implementation on transputers [15][16][17][18][20] which was the target architecture of the system.

To take advantage of the parallel query executor, an extension to the conventional database management system structure was proposed. The extended structure contains the conventional one, so conventional query processing is possible without the users noticing any change. The extended structure clearly separates the responsibilities of the parallel OFL executor and the rest of the system (data access module), this allowed us to concentrate on the key issues throughout the development phase. The extended database machine is a MIMD architecture, the processors are arranged in a pipeline and grouped together

into functional units. Each functional unit is assigned a portion of the program and data are passed from one unit to the next during execution, that is the problem was solved by functional decomposition.

The execution of an OFL program consists of two phases. In the first, preprocessing phase, the structure of the program is analysed and the functional decomposition takes place. The separated program fragments are assigned to the functional units of the extended database management system and translated into abstract machine code. In the second phase, the translated program fragments are downloaded to the functional units and executed. The executor itself is a multiprocessor implementation of the OFL abstract machine written in parallel C.

2.2 OFL specification

OFL programs are functional expressions and manipulate *objects* via *function* calls.

2.2.1 Objects

Typically some objects contain other objects as their members, these are called *collections*. Every collection must support three functions for the purpose of enumerating their members (see below). Apart from three *special objects*, the OFL executor does not recognize the identity of the objects. These special objects are:

- ▶ boolean *false*
- ▶ boolean *true*
- ▶ the object *nil*, used by the built-in functions

2.2.2 Object references

In OFL programs an *object reference* is used to access an object. There are two kinds of references:

indirect reference by *object variables*, these are either

- ▶ *false*, *true*, *nil* referencing the special objects mentioned above, or
- ▶ a *user variable*, defined by using the *assign* built-in function (see below).

direct reference to objects, these are string literals to the OFL executor. It is the responsibility of the function to which these direct references are passed to determine the identity of the object described by the string literal.

2.2.3 Functions

Every function accepts objects as its parameters and passes back a single object as its result, ie. functions are single-valued and 'object' is the only data type in OFL programs. The same function name can be used to denote several functions with different arities. A parameter (argument) to a function can be an object reference or a function call. There are two kinds of functions: *built-in* and *external*.

External functions are not carried out by the OFL executor itself, their arguments are determined and passed—along with the name of the function—to the *environment* in which the executor is run. That environment is responsible for evaluating the function and returning the result back to the executor. There are *mandatory (traversal)* and *optional (behavioural)* external functions. Each collection maintains a pointer to its *current member* internally, this pointer can be set and moved by the traversal functions, this way the collection members can be enumerated.

Traversal functions to enumerate collection members:

- ▶ `Current( argument )`
Returns the current member of the collection identified by *argument*.
- ▶ `First( argument )`
Sets the internal pointer to the first member and returns *true* if the collection is not empty. Returns *false* if the collection is empty.
- ▶ `Next( argument )`
Returns *false* if all members have been enumerated. Sets the pointer to the next member of the collection and returns *true* otherwise.

Behavioural functions are any external functions that appear in an OFL program and are not traversal functions.

Built-in functions are predefined by the language:

Here a predicate is either *false*, *true* or a function that returns *false* or *true*. All built-in functions return *nil*.

- ▶ `assign( object_reference, argument )`
object_reference is treated as an indirect object reference throughout the program. The object it identifies is determined by the argument.
- ▶ `sequence( argument1, argument2, ..., argumentn )`
Evaluates the *arguments* in the given order, which are normally functions. The values are not used.
- ▶ `while( predicate, function )`
Calls the *function* until the *predicate* becomes *false*. Each iteration begins with the *predicate* evaluation.
- ▶ `if( predicate, function1, function2 )`
If the *predicate* returns *true*, *function₁* is called. If the *predicate* returns *false*, *function₂* is called. If the first parameter is not a predicate, the behaviour is undefined.
- ▶ `forany( object_reference, predicate, function )`
Enumerates the members of the collection identified by *object_reference* until the value of *predicate* becomes *true*. Then the *function* is called. If the collection is empty or *predicate* is never *true*, the *function* is not called.
- ▶ `forall( object_reference, predicate, function )`
As *forany*, but enumerates all members of the collection and *function* is called each time *predicate* is *true*.

An OFL program is a (compound) function, it is executed by traversing the function tree. Every function evaluates its arguments before computing its result.

2.2.4 OFL grammar

Based on the above specification, the context-free grammar of OFL can be defined as follows:

```

ofl = function;

function = external;
function = built-in;

external = "Current" "(" objectref ")";
external = "First" "(" argument ")";
external = "Next" "(" argument ")";
external = Functionname "(" arguments ")";

arguments = ;
arguments = argumentlist;

argumentlist = argument argumentlistrem;

argumentlistrem = ;
argumentlistrem = "," argumentlist;

argument = function;
argument = objectref;

objectref = Identifier;

built-in = "assign" "(" objectref "," argument ")";
built-in = "sequence" "(" arguments ")";
built-in = "while" "(" predicate "," function ")";
built-in = "if" "(" predicate "," iftrue "," iffalses ")";
built-in = "forany" "(" objectref "," predicate "," function ")";
built-in = "forall" "(" objectref "," predicate "," function ")";

iftrue = argument;
iffalses = argument;
predicate = argument;

```

Implementation note (see Section 2.4): Functionname in the present implementation begins with an uppercase letter followed by any number of letters, numbers or underscores. Identifier in the present implementation can be quoted or unquoted. If unquoted, it cannot begin with an uppercase letter as this format is reserved for function names, otherwise its format is the same. If quoted, it begins and ends with a quote (') and in between can contain any number of characters that are not newline or quote. A quote can be put in as \' and a backslash can be put in as \\\.

2.2.5 Database management with OFL

A conventional database management system can be depicted as in Figure 2.1. User queries are transformed and optimised before execution by a query preprocessor. The database is accessed and manipulated by a Database Manager (DBM).

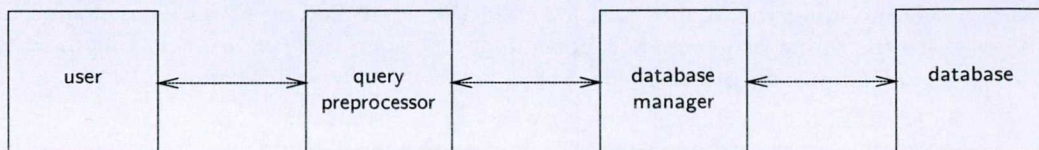


Figure 2.1: Conventional database management system

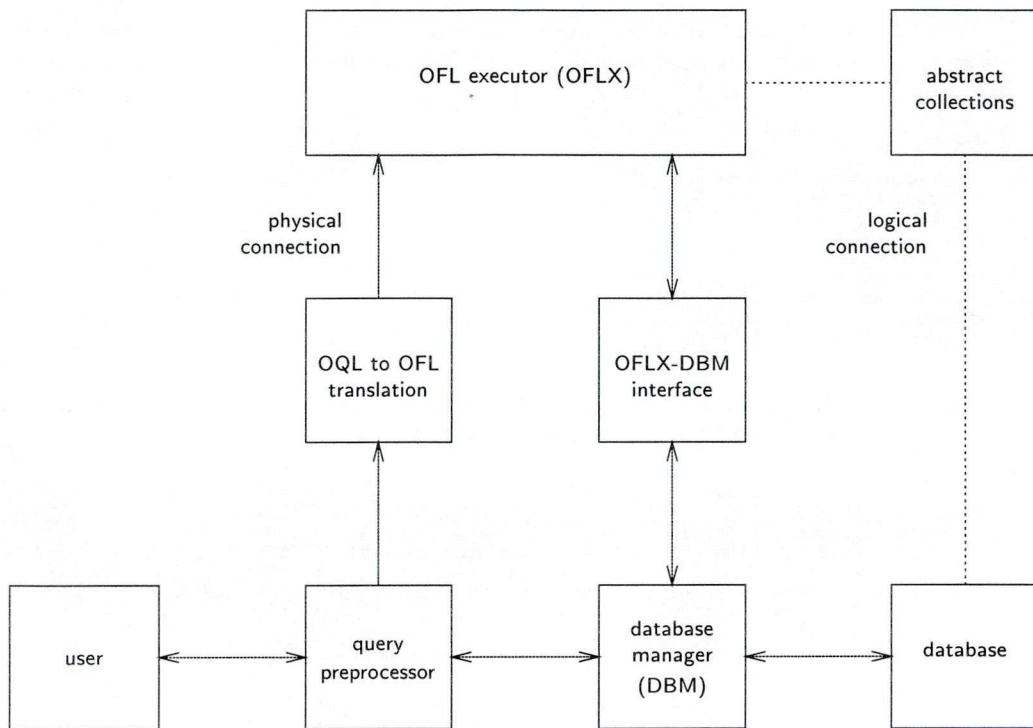


Figure 2.2: OFL DBMS

A Database Management System completed with the OFL system is shown in Figure 2.2. As OFL is object oriented, it is supposed that the user issues queries expressed in OQL. In the query preprocessing phase, the OQL query is translated into OFL. During the execution of the OFL program the OFL executor passes messages to the OFLX-DBM interface unit. The messages describe external function calls to be carried out by the interface unit, which in turn passes back the result of the function. Database management in our system consists of the following steps:

1. **Description of a database in ODL.** The ODL description is transformed into an internal representation for subsequent use. For the present purposes, simple relational databases are used with a couple of tables. A simple database in ODL:

```

interface People (extent people) {
    attribute String firstname ;
    attribute String lastname ;
    attribute Short age ;
    attribute String owner ;
} ;

interface Cars (extent cars) {
    attribute String carcode ;
    attribute Short composed ;
    attribute String color ;
} ;

interface CarParts (extent carparts) {
    attribute Short partcode ;
    attribute String label ;
} ;
  
```


2. **Assembly of OQL queries.** At present, queries are restricted in the sense that they can not contain update operations. A query for a database of the structure above:

```
select
  struct(
    lastname: p.lastname,
    color:    v.color,
    label:    c.label)
from
  p in People,
  v in p.owner,
  c in v.composed
where
  p.age = 16 ;
```

3. **OQL $\Rightarrow$ OFL translation.** Based on the language definition, a translator has been implemented. The OFL program equivalent to the former OQL query:

```
sequence(
  assign( _person, 'People'),
  forall( _person, IntEq( Field( 'age', Current(_person)), '16'),
    sequence(
      assign( _vehicle, Field( 'owner', Current(_person))),
      forall( _vehicle, true,
        sequence(
          assign( _part, Field( 'composed', Current(_vehicle))),
          forall( _part, true,
            sequence(
              Display( Field( 'lastname', Current(_person))),
              Display( Field( 'color', Current(_vehicle))),
              Display( Field( 'label', Current(_part)))
            )
          )
        )
      )
    )
  )
)
```

4. **OFL program preprocessing.** Before the execution of an OFL program it has to be analysed and preprocessed so that the executor would be able to evaluate it efficiently. OFL programs generated to be evaluated using the pipelined strategy have a common structure. This structure is best shown by the so-called abstract collection traversal graph, which is the last step in the generation sequence before an OFL program is produced. The forall function has the most important role in this respect, it determines the overall structure of a query.

The preprocessor has to look for the first forall function occurrence in the OFL program and split up the OFL tree (program) so that the subtree corresponding to the third argument of the forall is separated. This is performed iteratively until the resulting subtree contains no forall. The tree segments found this way will be evaluated by the functional units introduced in Figure 2.4.

The example program given earlier is sliced as shown below. The program is separated into four functional units.

```
► sequence(
  assign( _person, 'People'),
  forall(
    _person,
    IntEq( Field( 'age', Current( _person)), '16'),
    *
  )
)
```



```

▶ sequence(
  assign( _vehicle, Field( 'owner', Current( _person))),
  forall(
    _vehicle,
    true,
    *
  )
)
▶ sequence(
  assign( _part, Field( 'composed', Current( _vehicle))),
  forall(
    _part,
    true,
    *
  )
)
▶ sequence(
  Display( Field( 'lastname', Current( _person))),
  Display( Field( 'color', Current( _vehicle))),
  Display( Field( 'label', Current( _part)))
)

```

5. **OFL parallel execution.** The evaluation proceeds in two steps. First the program segments are distributed to the functional units. In the second step, evaluation of the segments is performed in parallel. Functional units can proceed unless an object is available to work on. Using a TNode transputer-based machine, a parallel database machine was modelled and the OFL program executed. During execution, runtime data is gathered for subsequent analysis.
6. **Graphical analysis** of runtime execution data. A graphical tool has been designed and implemented to allow runtime data to be presented and analysed in a convenient manner.

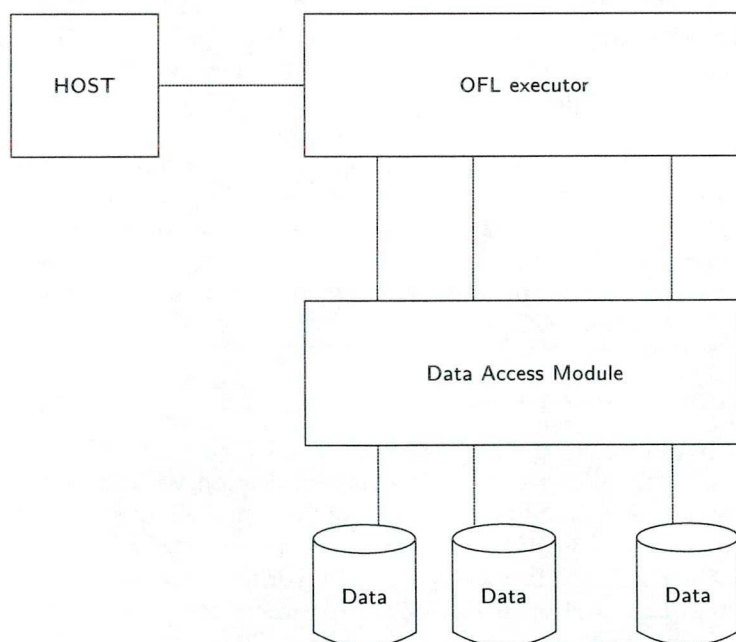


Figure 2.3: A parallel database machine

2.3 Parallel executor for OFL programs

For the parallel execution of OFL programs a generic parallel database machine architecture is proposed. The machine is based on the MIMD model of parallelism, this allows the system to be built from transputers.

2.3.1 Parallel Database Machine

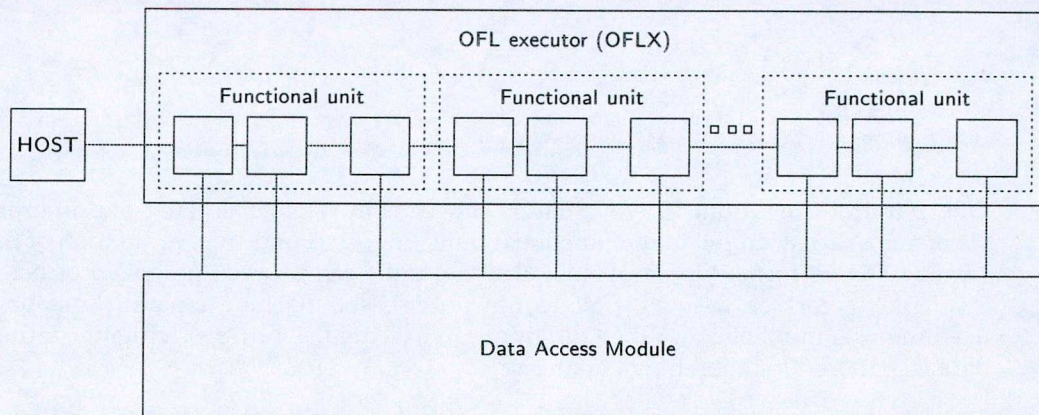


Figure 2.4: Executor internal structure

The database machine is separated into two units, the *executor* and the *data access module* (Figure 2.3). Both are assumed to consist of several processors. The task of the data access module is to serve the executor. It encompasses both the OFLX-DBMS interface and DBMS modules (see Figure 2.2). We do not deal with the internal structure of the access module here, it is assumed that it carries out its assigned tasks. With this assumption we can concentrate on the execution of OFL programs, while realizing that the inner organisation of the access module is not trivial. The system is connected to a host machine where query input and query preprocessing takes place. The OFL executor uses the pipelined evaluation strategy to process queries. For this reason, the processors in the executor are connected to each other in a pipeline (Figure 2.4).

2.3.2 Functional unit structure

A functional unit consists of the processes shown in Figure 2.5. The input and output processes are responsible for handling the communication with the neighbouring units. The buffer process stores data that are results of the execution of the program assigned to this particular functional unit. If the neighbour on the right becomes idle, it retrieves data from the buffer and continues its processing with working on the new piece of data.

The most important part of this process structure is the *evaluator* which is an OFL abstract machine and executes the same compiled OFL code repeatedly after every data packet received. Remember that the code assigned to the evaluator is determined in the OFL preprocessing phase.

We have not taken into account so far how many processors we have for the executor. If we have at most as many processors as functional units, then the scheme above is appro-

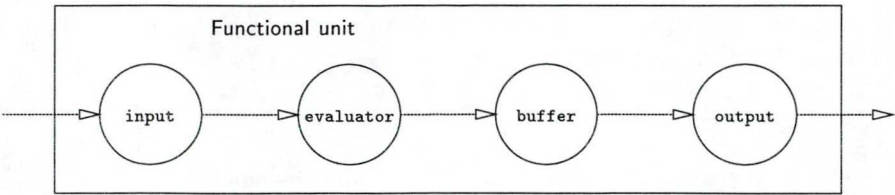


Figure 2.5: Single processor functional unit

appropriate. But as the number of functional units is determined by the number of `forall`s in a program and that number is limited, we should consider the case when there are more processors than functional units. In this case we assign several processors to a single functional unit, but this induces minor changes in the process structure as shown in Figure 2.6.

It can be seen that the input, output and evaluator processes are repeated as many times as necessary, but still there is only one buffer process. Further the input and output processes of a processor are connected to be able to communicate with each other directly. This is so because this way it is possible for all evaluators to access new data via a chain of input and output processes.

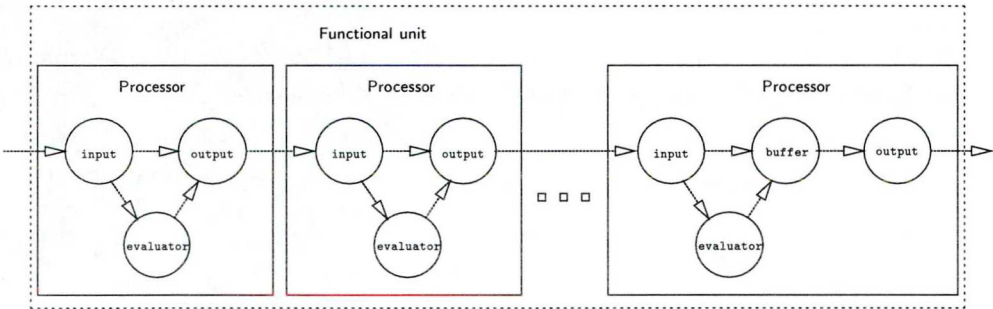


Figure 2.6: Multiprocessor functional unit

2.3.3 OFL abstract machine

In order to define the semantics of OFL programs and to ease execution, an abstract machine was defined. The machine consists of a *code vector*, an *object stack*, an *object table*, a *function table* and of course the *control logic*.

The object stack is used to store objects that are parameters of functions to call. There is also a special stack object called *mark* that is used to mark stack positions. The *mark* is different from any other object. The object table stores the user variable bindings during execution and the literal objects encountered during parsing the OFL program. (see the *load* instruction below). The function table contains the names of the external functions as they were encountered during program parsing. The code vector contains *instructions* for the machine. Any instruction can be addressed by its index in the code vector. The instructions are executed in sequential order unless one of the special jump instructions is encountered. Most instructions examine or alter the state of the stack or the object table,

all of these effects are going to be described. An instruction has two parts, the instruction code and the instruction argument.

The following instructions are available:

- ▶ `jump instruction_index`
Control is transferred unconditionally to the instruction at code vector slot *instruction_index*. It has no other effects.
- ▶ `jtrue instruction_index`
Control is transferred to the instruction at code vector slot *instruction_index*, if the topmost element of the stack is the object *true*. It has no effect on the execution order if the topmost element is *false*. In either case the topmost stack element is removed. If there is any other object on the top of the stack or it is empty, the effect of the instruction is undefined.
- ▶ `jfalse instruction_index`
Control is transferred to the instruction at code vector slot *instruction_index*, if the topmost element of the stack is the object *false*. It has no effect on the execution order if the topmost element is *true*. In either case the topmost stack element is removed. If there is any other object on the top of the stack or it is empty, the effect of the instruction is undefined.
- ▶ `load table_index`
This instruction pushes a new object onto the top of the stack. It copies that object from the object table from the slot identified by *table_index*.
- ▶ `assign table_index`
This instruction removes an object from the top of the stack. It copies that object to the object table into the slot identified by *table_index*. If the topmost element on the stack is *mark*, or the stack is empty the effect of the instruction is undefined.
- ▶ `call function_index`
It calls an external function. The name of the function is stored in the function table at slot *function_index*. The parameters are assumed to be on the top of the stack. A *mark* special stack object is used to mark the place where the instruction should stop reading the parameters. All parameters including the mark are removed from the stack as a result of this instruction. After the function is executed, its result is placed on the top of the stack.
- ▶ `send table_index`
This instruction is specifically designed for the purpose of execution in a multiprocessor system. It duplicates the object table, executes an `assign table_index` instruction using the duplicate table and sends that table on a message channel to a neighbouring processor (see the translation of the `forall` function).
- ▶ `oper operation_code`
This instruction is used to encode operations that do not require parameters. For each different value of *operation_code*, it encodes a different operation.

At present the following three operations are available:

- `mark`
Pushes a *mark* object onto the stack.

- **pop**
Discards the topmost element of the stack. Its effect is undefined if the stack is empty.
- **clear**
Discards all elements of the stack from the top to (and including) the topmost *mark*. If there is no *mark* on the stack, all elements are discarded.

2.3.4 Preprocessing of OFL programs

It has been discussed earlier that a preprocessing phase is required before OFL programs can be executed by the parallel database machine.

The preprocessing encompasses the following steps:

syntax analysis of the OFL program and building of a parse tree that is the basis of further processing. In the current implementation the PROF-LP system [54] is used for syntax analysis. This system is a compiler generator tool that allows us to build a compiler from an attribute grammar specification. The grammar used is the one listed in section 2.2.4 and the semantic functions have been designed so as to their result is a parse tree in memory. This parse tree can be used for further processing.

selection of forall functions to determine how many functional units are needed and which functions are assigned to them. The forall functions selected for code separation purposes all lie on a path starting from the root of the parse tree. This path is called the *backbone* of the tree for easier reference. In principle any such path of the parse tree could be the backbone but not all of them can be expected to be equally preferable. More work is needed to find out what factors should be considered when the backbone is determined.

translation of the OFL source code into abstract machine code for each functional unit.

Details of the translation process are described in Section 2.4.1. Throughout the following description $\text{index}(\text{identifier})$ stands for the slot index in which *identifier* resides in the appropriate table and $\text{code}[\text{syntactic_unit}]$ denotes the OFL abstract machine code assigned to *syntactic_unit*.

```
code[objectref] =
    load    index(objectref)

code[ arg1, arg2, ..., argn ] =
    code[arg1]
    code[arg2]
    ...
    code[argn]

code[external_function( argument_list )] =
    mark
    code[ argument_list ]
    call    index(external_function)

code[assign( objectref, argument )] =
    code[ argument ]
    assign  index(objectref)
    load    index(nil)
```



```

code[sequence( argument_list)] =
    mark
    code[argument_list]
    clear
    load    index(nil)

code[while( predicate, function)] =
    jump    TEST
    AGAIN:  code[function]
    pop
    TEST:   code[predicate]
    jtrue   AGAIN
    load    index(nil)

code[if( predicate, iftrue, iffalse)] =
    code[predicate]
    jfalse  FALSE
    code[iftrue]
    jump    OUT
    FALSE:  code[iffalse]
    OUT:    pop
    load    index(nil)

code[forany( objectref, predicate, function)] =
    mark
    load    index(objectref)
    call    index("First")
    jfalse  END
    jump    TEST
    NEXT:   mark
    load    index(objectref)
    call    index("Next")
    jfalse  END
    TEST:   code[predicate]
    jfalse  NEXT
    code[function]
    pop
    END:    load    index(nil)

code[forall( objectref, predicate, function)] =
    mark
    load    index(objectref)
    call    index("First")
    jfalse  END
    BODY:   code[predicate]
    jfalse  NEXT
    forall_body
    NEXT:   mark
    load    index(objectref)
    call    index("Next")
    jtrue   BODY
    END:    load    index(nil)

```


In the last listing *forall_body* stands for different portions of code depending on whether the *forall* function in question is part of the backbone or not. The simpler case is when it is not part of the backbone: `code[forall_body] =`

```
code[function]
pop
```

If the *forall* is a member of the backbone: `code[forall_body] =`

```
mark
load    index(objectref)
call    index("Current")
send    index(objectref)
```

This code segment retrieves the current member of the appropriate collection (identified by *objectref*), creates a copy of the object table, in the copy replaces the collection with its current member and sends off the copy table. That table when received by the next functional unit is used as a starting object table.

Care is needed when `Current( objectref)` appears inside a *forall* that is contained in the backbone and *objectref* is the first parameter of the *forall*. As a result of the execution of the code segment above the current member of the collection can be found in the slot reserved for the collection itself. For this reason the *Current* function is not invoked but a single *load* instruction is inserted instead (see also Section 2.4.1).

2.4 Implementation

Appendix A contains the source code of the programs that were written to implement OFL. There are two programs, one to translate OFL programs into OFL abstract machine code (Appendix A.1), the other to execute OFL abstract machine programs along with number of supplementary modules needed to create an environment in which OFL programs can operate (Appendix A.2). Source lines will be identified with their sequence number appearing in the listing (eg. 123).

2.4.1 OFL Translator

The translator was written in C, to compile the programs we used the Microsoft C compiler, version 6.00. The `makefile` (1) contains the rules to build the translator from its source components. The `std.h` (67) file contains some global declarations used in the program.

Parser

To generate the parser, the PROF-LP compiler generator system [54] has to be present, it is a compiler generator system that accepts a language description in an attribute grammar form [55][56]. Although it would be capable of producing a compiler to compile OFL programs, that compiler would produce sequential C code, so this approach is not suitable for the present purposes. A more favourable approach would be to describe an attribute grammar for a compiler that produces OFL abstract machine code. This approach also has drawbacks which make it unattractive: most of the attributes would be large tables, whose handling requires a lot of programming, so we decided to have PROF-LP generate a "compiler" whose only task is to produce a derivation tree of the OFL source. This derivation tree is then subjected to further analysis to collect the necessary information into tables and produce the OFL abstract machine code.

The translation works as follows. The syntax and semantic functions to create a derivation tree of an OFL program is described in a form acceptable to the PROF-LP generator. `ofl.lxd` (□80) contains the lexical description needed for lexical analysis, `ofl.hlp` (□112) contains the grammar augmented with the semantic equations which call tree-constructing functions from `oflparse.c`. Attribute `pt` (□115) is “parse tree”, `ptl` (□116) is “parse tree list”.

The file `oflparse.h` (□267) contains the declarations of structures used to build the parse tree. `ParseTreeNode` (□275) is the type of a node in the tree and has five fields:

- ▶ `NodeAttrib` stores the attribute of the node, it can contain one of the `NATTR_*` values defined between lines □291–□311. `NATTR_INTERNAL_FORALL_X`, `NATTR_OBJREF_ASSIGNED` and `NATTR_OBJREF_FORALL_X` are used only in the second phase, after the derivation tree has been constructed.
- ▶ `NodeLabel` is a string containing the label of the node, for example the name of an external function or the characters of a user variable.
- ▶ `NofParams` contains the number of subnodes a node has. For example, the internal function `if` has three parameters, `NofParams` has value 3 in that case.
- ▶ `info` is filled with `NULL` initially, it is used later to hold a pointer to the generated code associated with the node.
- ▶ `Param` is a pointer to a dynamically allocated array of nodes (node pointers), these are the subnodes of a particular node. Its size is stored in `NofParams`.

`PTListItem` (□285) is used to construct lists of nodes corresponding to argument lists of functions.

The global `ParseTree` (□314) variable will contain the pointer to the root of the parse tree, the function `oflx` is invoked after the parse tree is constructed to continue processing.

The file `oflparse.c` (□319) contains the functions which are called by the PROF-LP generated code at appropriate times during the parsing of an OFL source program. By inspecting `ofl.hlp`, we can see that the functions `AddParam` (□406), `AddPTNodeList` (□392), `AddPTNode` (□377) and `ofl_close` (□416) are present in the semantic functions of the attribute grammar. Two other auxiliary functions are used internally, `CheckPtr` (□329) is used to check for out of memory conditions after dynamic memory allocation, `ProcessString` (□337) converts an identifier to internal form by removing escape characters from the name.

Code generator

When the parse tree has been constructed, the function `ofl_close` is called. It checks whether any syntax error occurred and calls `oflx` (□1091) to continue the processing. It carries out the following steps:

- ▶ Backbone determination (□1096).
- ▶ Compiling the object table and the function table (□1097–□1098).
- ▶ Generating OFL abstract machine code (□1100–□1102).
- ▶ Creating an object file containing a header, tables and code (□1104–□1135).

The variables and data structures used in this phase are declared at the beginning of `oflx.c` (□468–□552). Three tables are used internally, a function table and two object tables, one for direct objects and another for indirect ones. These two are finally merged into a single object table by the `FillTables` function. All these tables have a common structure, an entry is defined by the type `tableentry` (□470) containing a string and an attribute. A table descriptor `table_desc` (□475) identifies a particular table by storing a pointer to the first table entry (`tableptr`), the maximum number of entries (`tablesize`) and the number of entries used (`nof_entries`). Next, three tables and their descriptors are defined, `did_table/did_table_desc` for direct objects, `iid_table/iid_table_desc` for indirect objects and `fnid_table/fnid_table_desc` for functions. `Did_table` (□485) is initialised to contain the names of the three predefined objects *false*, *true* and *nil*. Although references to these are indirect, their treatment is special in that they are always put into the first three slots of the final object table. `Fnid_table` (□509) is initialised to contain the names of the three mandatory traversal functions *Current*, *First* and *Next*. The variable `ind_table_index` (□519) is set to the index of the first indirect object after the two object tables are joined.

The structure `instruction` (□525) stores OFL abstract machine instructions. Each instruction has a token (instruction code), an arg (instruction argument) and a rem (remark) field. The `#included ginstr.h` file (□428) contains the instruction codes (and names for debugging) for the abstract machine. This file is shared between the translator and the executor. The structure `blockentry` (□535) is used to store either an instruction or a block of code. The block structure (□543) declares the type for code blocks, it is an array of block entries, that is a sequence of instructions and/or blocks. The `blength` (block length) field stores the size of the `bvector` array, the `tlength` (total length) field contains the number of instructions in this block. The number of instructions can be greater than the length of the block if the block contains subblocks. The variable `code` (□551) is an array of code blocks, each code block will be assigned to a separate functional unit by the executor. `Codeidx` (□552) stores the index of the code array which is being filled.

Two functions associated with table handling are `Lookup` (□555) and `Insert` (□569) with obvious intent. If we want to insert an existing item into a table, then the attribute of that item is updated.

Next come the functions which `oflx` calls in order to carry out its task.

`BackBone` (□589) determines the backbone of the parse tree, that is it selects and marks those `forall` functions that will be assigned to different functional units. It does this by changing the attribute of these nodes from `NATTR_INTERNAL_FORALL` to `NATTR_INTERNAL_FORALL_X`. At present the algorithm applied works as follows: Given a parse tree, the upper rightmost `forall` is detected and the subroutine is invoked recursively for the subtree determined by the third parameter. The `foralls` encountered this way constitute the backbone.

The function `FillTables` (□607) traverses the tree and extracts information about functions and objects into the corresponding tables. Special attention is devoted to `NATTR_INTERNAL_FORALL_X` and `NATTR_EXTERNAL_CURRENT` attributes. If the object variable in the parameter of a `Current` function is also the first parameter of a `forall` in the backbone, then the attribute of that particular `Current` is also changed to `NATTR_OBJREF_FORALL_X`. This prepares for the pipelined execution of the program (see also the `send` abstract machine instruction).

The function `JoinTables` (□666) creates a single object table from the two temporary ones and sets up the variable `ind_table_index` to mark the first indirect object.

The subsequent functions are used during code generation. `AllocInstr` (□683) allocates a single instruction structure, `AllocMARK` (□700) and `AllocLOAD` (□705) are spe-

cialised versions creating an `oper` mark and load instruction respectively. `AllocBlock` (715) allocates a block of the requested size for an instruction/block sequence.

`CodeForall` (724) generates code for a `forall` function, `CodeSend` (781) generates code to be incorporated into the code of `forall`s in the backbone. The fairly long but simple structured `CodeGen` (809) function generates code for every function calling the former ones where necessary. The function is a straightforward implementation of the definitions given in Section 2.3.4.

The last function, `CodeExtract` (1052), extracts the instruction sequences from the blocks and writes them both to the object file and to the standard output. The object file content produced by translating the example OFL program shown earlier is listed in Appendix A.3 (2613).

This concludes the OFL translator program, the generated object files are input by the executor, which is going to be described next.

2.4.2 OFL executor

The executor and its accompanying modules were written in INMOS C [48] for the transputer. The system was developed and runs on a Telmat T.Node machine. The `makefile` (1140) contains all the rules used to build the executor from the source files. The file `global.h` (1197) contains the manifest constant `DBM_BUFFER_SIZE` which defines the largest possible size of messages that can be sent between individual components. `Std.h` (1203) contains some global declarations used throughout the program.

Structure

In order to be possible for the executor to function, it has to be put in an environment described earlier in Section 2.2.5 and 2.3.1. The components in the environment have been implemented only to a degree which enables the executor to run. They were, however, designed as independent modules with properly defined interfaces, so they can be improved independently of the rest of the system.

The highest level description of a parallel program on a transputer network is the configuration file. It consists of three sections:

- ▶ Hardware network definition, *processors* are connected via communication *links*.
- ▶ Software network definition, *processes* are connected via communication *channels*.
- ▶ Mapping, processes are placed on processors and channels are placed on links.

The file `oflx.cfs` (1230) contains the configuration information for the executor environment. It is written in a special configuration language, but the configuration itself can more easily be understood by studying Figure 2.7. The construction can be extended to the right to accommodate more processors. The processor attached to the host contains the master process which mediates between the host machine and the rest of the system. The upper row of processors contain the processes of the database system (`oflx`: OFL executor, `dbmi`: database manager and interface). The processors in the lower row contain a special router process. These allow us to send debug messages to the console in runtime without interfering with the other processes. This is important because the executor processes collect runtime performance data and these would not be accurate if the executor was interrupted to handle debug messages.

The router process (1362) routes messages to the master process. Each has one output channel leading to the next router in the row and three input channels coming from

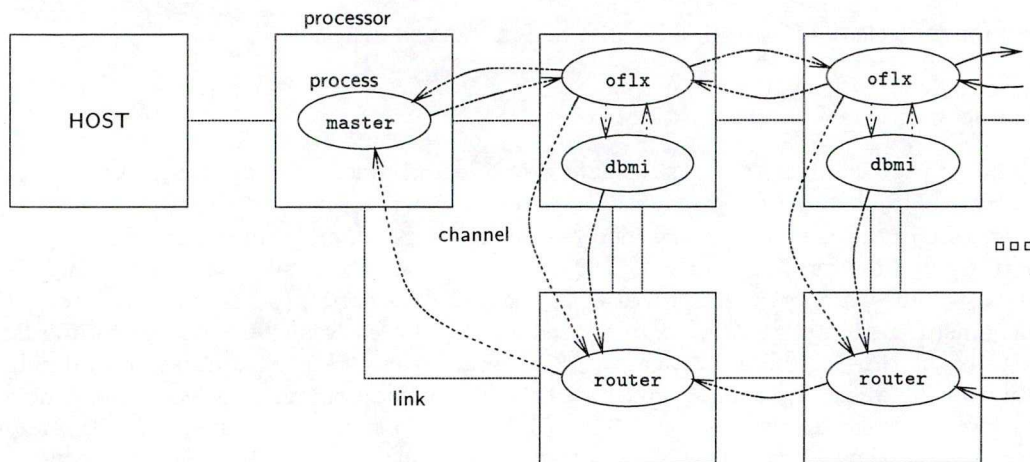


Figure 2.7: OFL executor environment configuration

the previous router and from the `oflx` and `dbmi` processes above it, respectively. The router is implemented as a standard producer-consumer component, this method is well known and widely used for this type of task. It consists of three processes. `Prod` (1385) receives messages from the previous router, this way it “produces” the messages. `Buff` (1404) stores the incoming messages until they can be passed over. `Cons` (1441) takes the messages from `buff` one by one and passes them to the next router, it “consumes” the messages.

The master process (`main`) plays the role of the HOST component of Figure 2.3. When the system is started, the user has to supply two file names as command line parameters. The `main` function checks these. The first file should contain a list of file names, each of which containing a relational database table in a simple text file format. The master process distributes the contents of these files line by line to the `dbmi` processes by calling the function `load_dbase`. The second file on the command line too contains a list of files, in this case those files should be object files produced by the OFL translator described in the previous section. All OFL programs in the list are downloaded and executed in the `execute` function before the system terminates.

Execute contains a loop (1600) whose body is responsible to execute one OFL program. It checks whether the file is a valid object file (1610), then reads the header and the program, then passes all information to the evaluators by communicating on the appropriate channel, out_oflx. After the object file has been processed, it goes into a passive state by calling the displays function (1652).

Displays (1506; with hindsight, not the best named function) has two tasks. One is to relay the incoming debug messages to the console, the other is to determine whether the computation has finished. For this to be possible, it receives the number of activated functional units when it is called (parameter `quits`). Upon termination, each functional unit sends a special zero-length message signalling to the master that it has finished. The `displays` function counts the termination signals and returns if the counter (`nof_quits`) reaches the number of functional units.

The rest of the `execute` function collects the runtime data accumulated at the functional units and writes them to files before it continues with the next OFL program. The method



used for runtime data collection is described in the next chapter.

Abstract machine

The file `object.h` (1728) defines the data structure and constants for handling OFL objects. `oflx.h` (1745) declares global variables shared between `oflx.c` and `oflxs.c`.

The file `oflxs.c` (1786) contains three functions used before and after OFL programs are run. The function `load_dbase` (1792) reads in the database tables and sends them further down the chain of `oflx` processes and also to the `dbmi` process it belongs to. This mechanism ensures that all `dbmi` processes will have a copy of the database by the time programs start using them. `Of1start` (1820) receives an OFL program from its left neighbour and initialises the abstract machine components accordingly. Object table, function table and code vector are prepared when this function returns. `Of1stop` (1893) releases the memory taken up by the abstract machine data structures and sends off the runtime data collected.

The `oflx.c` (1925) contains the executor itself. At the beginning, there are a few definitions of variables used throughout the program. The variable `id` and the channels `inp_*`, `out_*` and `router` (1936–1940) are used to store the interface information of the process. The `record` (1968) and `diff_time` (1978) functions take part in runtime data collection. The function `debug_msg` (1996) is called to send a debug message to the console. The message string is prefixed with the `id` of the process and sent to the router, which forwards it to the master, which prints the text on the console.

The function `oflbuffer` (2007) contains the code of the buffering process (cf. Figure 2.5) which is executed in parallel with the evaluator. Its task is to temporarily store those object tables which result from executing the `send` instruction. As the present implementation assigns one processor to every execution unit and the moving of tables from one unit to the other can be achieved by a pair of messages, there is no need to launch separate input and output processes.

We now arrived at the abstract machine itself. The array `stack` (2076) is used as the object stack, `sp` is the stack pointer and `ip` is the instruction pointer indexing the code vector. The `pop` (2079) and `push` (2088) functions are the stack operations. `Sindex` (2097) searches the stack from the top for a specific object, it is used in the implementation of the `call` and `oper clear` instructions to look for a *mark*.

The function `am` (abstract machine, 2106) is called to execute the code loaded previously. It writes the three special objects into the predefined slots in the object table (2112–2114), then determines whether it should enter the main loop (2118) by trying to acquire an object table from the previous functional unit. When it enters the main loop, it acquires the table (only the objects above `IndirectIndex` are sent, because those are the ones that can change). Next, it executes its assigned code in a loop (2130–2313), where each instruction is handled in one of the alternatives of a `switch` statement.

At the main function (2362) we can see how the pieces are put together, as many OFL programs are executed as arrive on the input channel.

Database access

The process `dbmi` in Figure 2.7 contains both the OFLX-DBM interface and the DBM module from Figure 2.2. The task of the interface module (`dbmi.h` at 2391; `dbmi.c` at 2403) is to accept a function call (function name and parameters sent on a channel) from the executor, have the database manager execute the function and pass back the result to the executor.

The database module (dbm.h at [2531]; dbm.c at [2540]) has to evaluate the functions provided to it based on the database contents. This part was not worked out entirely by the end of the LPD project term, so the file listed generates random answers to function calls.

2.4.3 Tests

A series of tests have been carried out to evaluate the concepts that appeared during our work in the LPD project. We aimed to bring together all the components that range from the ODL description of a database to the evaluation of trace graphs produced during the parallel evaluation of an OFL query.

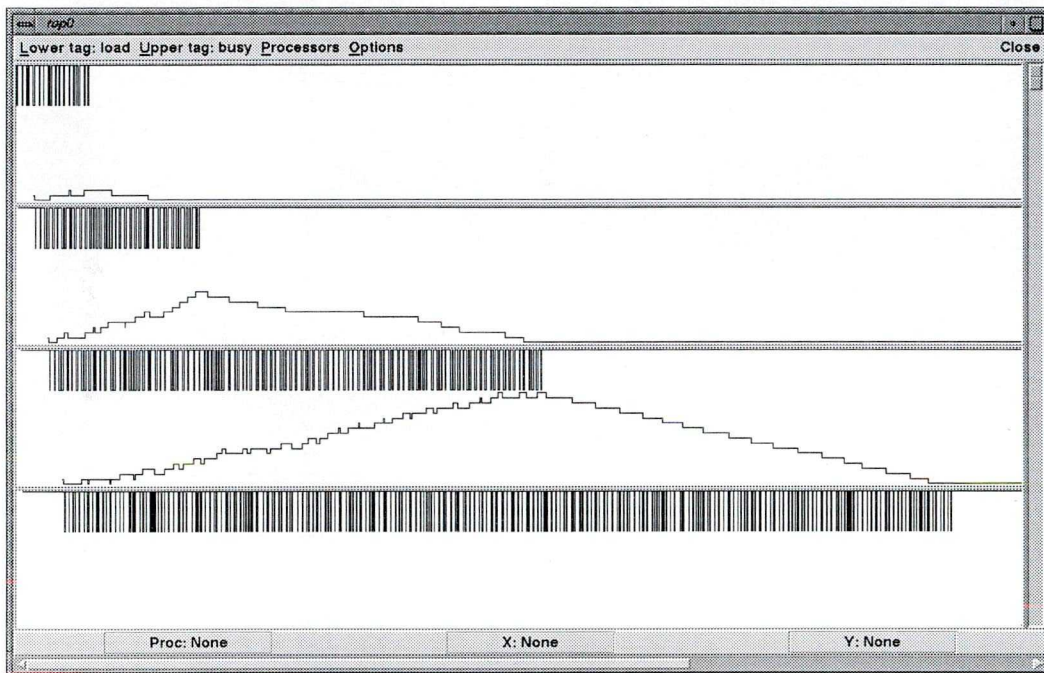


Figure 2.8: OFL executor performance graph

The graph in Figure 2.8 shows results from a typical test run involving four functional units. Each functional unit is represented by a horizontal band in the figure. During execution, two series of data have been collected. At each processor, the upper barcode-like region shows whether the control is at the OFL executor (down) or at the database interface module (up). The fine structure of this part can only be examined by zooming in on a particular segment with the graphical analyser program. The lower staircase-like region shows the load of the buffer of the respective processor. It can be seen that while a processor is working, it produces tasks for its right neighbour faster than it is capable of consuming, so the heap gets bigger. After a processor has finished, the heap gets smaller steadily as the neighbour processes the waiting tasks. One further improvement seems to be readily possible here: a processor that has finished with its own work should start consuming the tasks from its own buffer. For this to be possible, each functional unit has to be equipped with the code of its right neighbour as well.

It is evident that the pipeline technique can be used to speed up the execution of an OFL program and it is reasonable to conclude that the parallel OFL executor is able to exploit at least some of the parallelism that arises from a query. Further investigation should determine what the characteristics of a typical query are and what the structure of the data access module should be in order that the system could work with maximum efficiency.

Chapter 3

Dynamic load balancing system

Based on [3][4][5][6].

3.1 Introduction

For distributed memory multiprocessor systems the employment of appropriate load balancing strategies is crucial to the performance of most applications. In general, the strategy which leads to an even spread of work throughout the processing nodes is likely to produce the best overall performance ratings. In some instances this is straightforward to achieve: if it is possible to assess the computational and communication loads before program execution, the program sub-tasks can be appropriately allocated to the processing nodes. However for many applications these loads are not determinable in advance and hence load balancing becomes a dynamic activity that runs concurrently with the application.

One standard approach to the management of dynamic load balancing is the employment of *processor farms* [49]: a *master* processing node is responsible for the farming out of the tasks to the *worker* nodes, each worker computes and returns the task results and is allocated new work. This allows the work load to be well partitioned, even in the case where the computational complexity, and hence the execution time, varies from task to task. The drawback with the processor farm approach is the reliance on a central node to act as load balancer. In large multiprocessor machines this provides a communication bottleneck: this is particularly acute when subtasks are created dynamically within the worker nodes. In this case, work allocation involves conveyance of the newly formed task description to the central node, followed by transfer to its appropriately designated worker node. For many applications it is therefore likely to be of benefit to employ some form of locally based inter-processor work scheduling mechanism.

Work at Sheffield Hallam University on parallel declarative languages for the implementation of knowledge based systems led to the development of a parallel logic interpreter based on OR parallelism [24]. The significance of this application for the purpose of the load balancing investigation lies in the fact that the computational patterns that the system gives rise to are typical of a large class of applications in the artificial intelligence and knowledge based systems fields. In these applications the program structure can be described by means of an execution tree; the evaluator (eg. the parallel logic interpreter) accepts a task, processes it and produces descendants of the input record. Program execution continues with evaluation of the descendant tasks, the parent having ceased to exist on completion of the task spawning operation. In these applications parallel execution

gives rise to a highly dynamic system. During a program run, not only are tasks dynamically created in a manner which is unpredictable at the start of run time, but the tasks themselves vary considerably in their individual execution times thus producing the most difficult scenario as far as work scheduling and load balancing are concerned.

In the subsequent sections, work on a load balancing system for *decomposable problems* is described. The load balancing system has been developed to provide an environment in which the means of mapping decomposable applications to distributed memory multiprocessor systems can be explored. The processor farm approach of designating one master node to perform the task of controlling work distribution has been replaced with a *processor commune* model: under this approach, work allocation is a community activity with each processing node combining master-worker functionality. Each processing element continues to execute work (ie. evaluate tasks) and store any newly created tasks for local evaluation unless a state of work load imbalance is recognised. Processors maintain information about their own work load and compare it with that of neighbouring processors at appropriate times during program execution. When predefined thresholds are reached, work is transferred from a heavily loaded node to a lighter one.

In order to support investigation of different load balancing strategies, a test environment has been developed and comprises the following components:

- ▶ application model generator
- ▶ application executor
- ▶ load balancing analyser

The application model generator allows the computational characteristics of a particular program to be captured in the form of an application “mimic”. This skeletal version, which provides a real time emulation of the application under consideration, is used as the basis for investigation of load balancing strategies with the application executor. This operates in the form of a local balancer process, installed on every processing element and works concurrently with the local application tasks. Because of the separation of load balancing and application functionality, the system can be used to explore the performance of different applications under a range of predefined work scheduling algorithms. The runtime data obtained are then passed to the analyser tools.

Results obtained using this toolset are presented: these show that good performance can be obtained under nearest neighbour scheduling approaches. The overheads associated with the employment of work allocation mechanism are low and the manner of task distribution ensures that the system operates with maximum available parallelism throughout the application execution.

3.2 Test environment

A test environment, comprising a set of tools has been developed to support the investigation of the interaction between the computational characteristics of an application and differing load balancing strategies. Our work considers decentralised load balancing strategies for a class of decomposable problems which can be thought of as problems solvable by a form of parallel divide and conquer technique. These can be found in parallel logic systems, branch and bound search algorithms and a range of other applications.

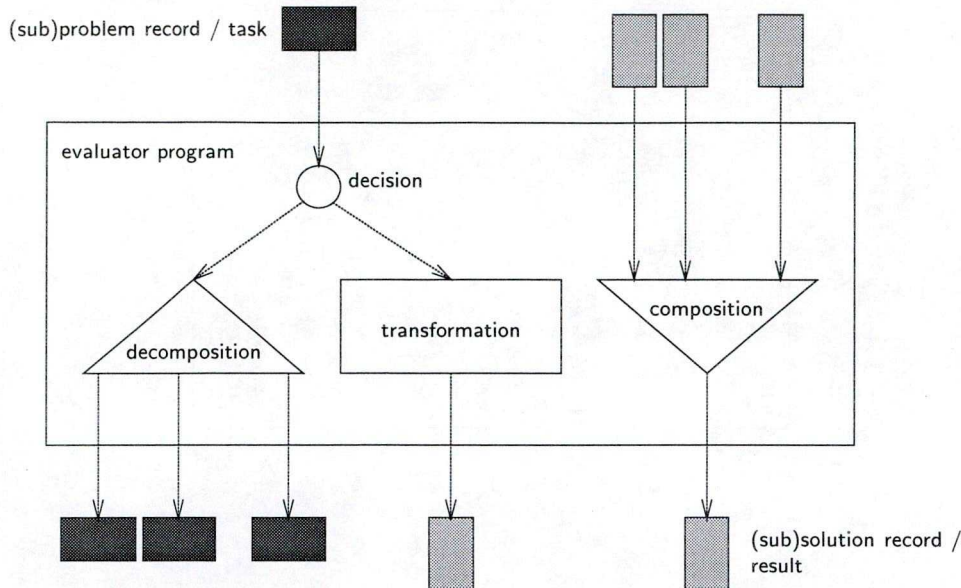


Figure 3.1: Generalised Decomposable Problem Schema

3.2.1 Decomposable applications

The process of solving these problems involves the successive *decomposition* into subproblems which is followed by a combination or *composition* phase and leads to the production of a required solution. In this model we assume that a *task* (the problem instance to be solved) can be encoded in a record which is then given to an *evaluator* which in turn produces the *result* or solution. These solutions can again be most easily stored in a (different) record.

In a decomposable application, in order to be able to speed up the computation, the problem input is expected to be decomposable so that the application program can—instead of producing a solution—split up the original problem into smaller pieces, each of which is encoded by the same schema as the original problem. The newly created subproblems thus become candidates for independent parallel evaluation. After the solutions for the subproblems are available, the program may wish to compose the subsolutions into a single record which encodes the solution to the original problem (Figure 3.1).

This way the application program processes a given task by applying one of the following operations:

- ▶ process an input record (task) and possibly produce an output record — *transformation* operation.
- ▶ process an input record and produce (several other) input records — *decomposition* or *spawning* operation
- ▶ process a group of output records and produce a single output record — *composition* operation

The operation chosen is influenced by the input the program is given, which is in turn determined by the environment. It is assumed that all the information needed for the com-

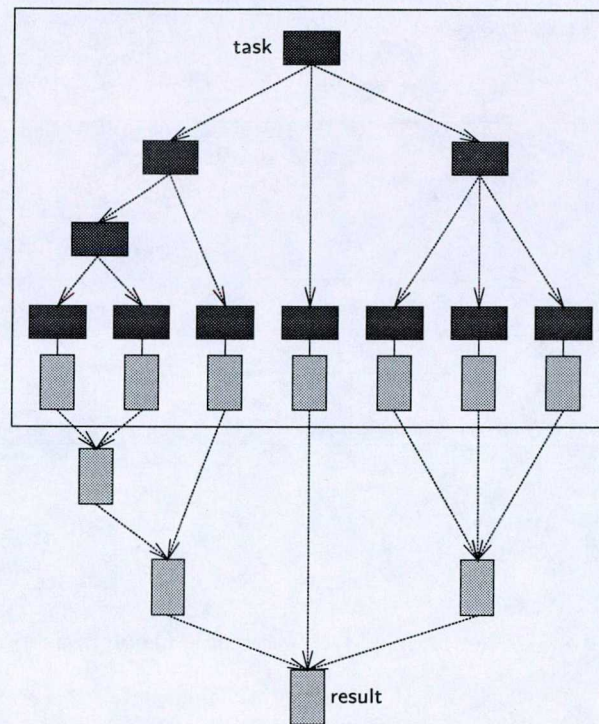


Figure 3.2: Decomposable Problem Execution Graph/Tree

putation is stored in the incoming records. This is necessary because in a parallel system in the course of the processing of a problem several program instances may be involved. Running on different processors, they have no easy way to share information. The way computation proceeds can be seen in Figure 3.2. Note that temporal relations are not shown between independent nodes.

We are going to deal with a somewhat simplified version of the model which we refer to as *special decomposable problems*. Here we assume that the composition phase of each program is a simple set-forming operation, ie. the solution for a problem is nothing more but the collection of subsolutions of subproblems into which it was decomposed. Thus the programs do not have to contain the code for the composition phase, the subsolutions obtained during the processing are either accumulated or returned one by one if some form of on-line processing is to take place. This form of processing leads to an evaluation tree rooted with the original problem, decomposed successively into subproblems and subsolutions obtained on the leaves of the tree (Figure 3.2, inside the rectangle). We also allow that a transformation does not produce a result. Borrowing terminology from logic languages, we say that the operation *failed* in such cases. Thus when dealing with special decomposable problems, three types of tasks have to be dealt with, depending on the outcome of their evaluation: *spawning*, *result* and *fail*.

3.2.2 Test environment components

The test modules provide a flexible system for testing the effect of varying different load balancing parameters on a range of special decomposable applications [4][5] (see Figure 3.3). Written in C and Parallel C [48], they originally ran on small IMB008 based transputer networks [15][16] and the PC host machine, but have subsequently been rewritten in OCCAM [45] and ported to a 16 processor T.Node [20][21] and Unix workstation.

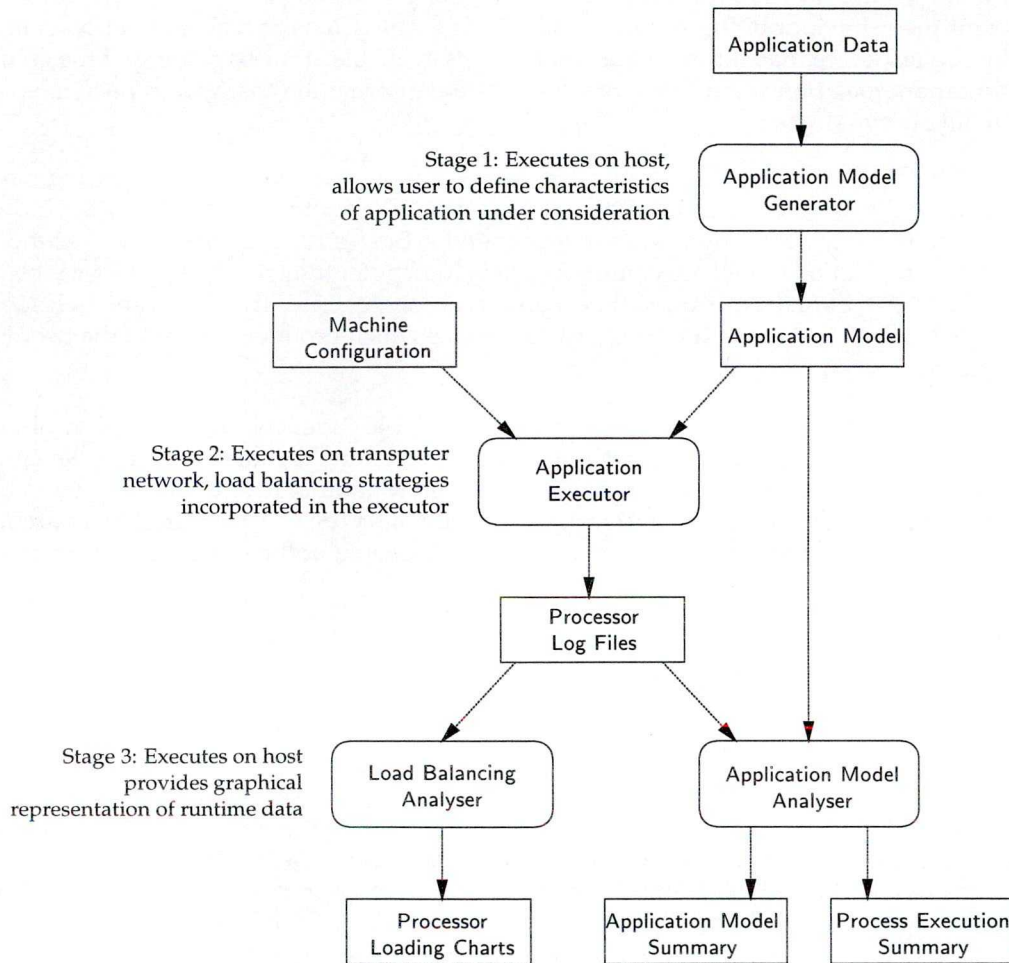


Figure 3.3: Test Environment

Stage 1: The application model generator produces an application model. By adjusting the input parameters to the generator, the characteristics of the target application can be set. The parameters include:

- ▶ the overall ratio of the number of different task types
- ▶ the range of execution times for different task types
- ▶ the range of record sizes required to hold task initialisation and result data

The generator creates application instances by generating encoded task trees by producing randomised values for the parameters, subject to the given range values. The encoded tree is then used as input to the application executor.

Stage 2: The application executor “executes” the application model and generates files of runtime data gathered from the processing elements in the network. The evaluator in the test system evaluates the application model by running idle loops in place of application tasks and producing empty data packets of sizes as prescribed. The runtime behaviour of the system is indistinguishable from the real application being modelled. Note that although the whole tree is available at the beginning of the computation, the application executor uses only the information that would be available in an on-line run.

Stage 3: The analyser modules then provide a detailed evaluation of the processing patterns and work loads during program execution. Two software tools have been designed to aid the analysis of the data generated in Stages 1 and 2. When the execution of the application model has completed, each transputer returns processor logs which include details of the execution times, processor loadings etc. These files are then subjected to data interpretation by the two modules which produce output to the user in a readily accessible form.

The significance of the stages in the test system is the independent generation of an application model which can be used repeatedly to evaluate a range of load balancing strategies on different machine configurations and allow comparative data to be derived. This synthetic representation of a decomposable problem has been constructed in an architecture independent manner and thus supports the extension of the test environment into different architectures.

3.3 Processor commune

3.3.1 Load balancing strategies

A number of different dynamic load balancing strategies are reported in the literature [59][60][61][62][63][64]. The benefit to be gained by transfer of work involves the evaluation of an appropriate trade-off between the accuracy of even work distribution and the overheads required to establish this even spread. Frequent monitoring of the state of the system has to be weighed against the proportion of time spent in the processing nodes performing load balancing operations, rather than application processing.

Work allocation can involve local or global information exchange operations, and be carried out in a synchronous or asynchronous manner. At one extreme, the traditional processor farm ensures that one processing node has exact information on the global state of work allocation at all times during execution: load balancing is globally controlled by asynchronous request-reply operations. In contrast, *diffusion* techniques rely on the node by node spread of work throughout the machine, with the assumption that under correct work exchange strategies, the load on each processing element will settle to an effective level. Thus control is completely decentralised and work transfer initiated between nearest neighbours by asynchronous communication.

Various intermediate load balancing strategies have been described: under a *hierarchical* balancing approach the processing elements are divided into a control hierarchy, and load balancing permeates up the hierarchy with controlling nodes centralising the process at

different levels. In the *dimension exchange* method [62] a global synchronous operation is performed in an iterative fashion by folding an N processing element network into $\log N$ dimensions and balancing each dimension consecutively. The *gradient method* [61] takes into account the proximity of processes in the determination of work transfer and a gradient map is maintained to capture this information.

The effectiveness of a load balancing strategy involves the following issues:

processor load evaluation: how to determine the load of a particular processor,

load balancing profitability determination: as data transfers have a high cost, it may be more profitable not to send even if there is an imbalance,

task migration strategy: which processors take part in the balancing action,

task selection strategy: which task(s) to send,

relationship among tasks: varying from independent to general precedence relations,

task granularity: the ratio of task execution time and their size; the coarser the granularity, the easier it is to balance the workload.

The evaluation of loads within each processing element and the identification of appropriate tasks to transfer if it proves advisable are dependent on the application; the determination of benefit and the recipient processing element are machine based issues. The benefit to be gained by transfer of work requires an evaluation of the degree of imbalance tolerated in the context of the communication/processing costs involved in relocation. It is therefore important to design load balancing systems so that these parameters can be evaluated and, if necessary, adjusted separately in order to provide a general purpose work allocation mechanism.

Experimental work on load balancing has considered these machine dependent aspects in relation to different allocation mechanisms [63]. Several assumptions about the nature of the application were made in order to simplify the investigation. These included the assumption that at start of execution there were a fixed number of tasks to evaluate, that these tasks could be executed in any order and were fully independent of each other, and that the load balancer would assume each task took one computation unit to process. The primary conclusion from this study was the superiority of diffusion techniques. However no direct comparison was made with the processor farm approach: in this situation farming of tasks should provide good performance because all tasks were defined at the start of execution and could therefore have been held centrally for eventual transfer to worker nodes.

In our study we have addressed the issues involved in diffusion load balancing strategies. Two approaches are possible: *sender initiated* work reallocation, in which an overloaded processing element (sender) requests relief from its neighbouring nodes and sends them work where appropriate; secondly, *receiver initiated* allocation — here the underloaded node (receiver) solicits work from its neighbours by request-reply operations. The mechanism studied in this project uses both methods.

The investigations have been driven by the more demanding nature of the application domain under consideration: it has been necessary to assume no pre-knowledge about the number of tasks to be executed, the time needed to execute them, or the amount of data required to initiate a transferred task on a neighbouring node. In addition, dynamic creation of tasks through decomposition operations means that inevitably some ordering of execution is imposed.

3.3.2 Commune load balancing strategy

A first approach to the mapping of a parallel declarative language system onto a transputer based multi-processor architecture involved a quasi-processor farm approach which was based round the broadcasting of process initialisation packets throughout the machine [25]. Because of the communication overheads involved in this approach, the design incorporated a broadcast bus to enhance the transputer link bandwidth.

However the development of the load balancing environment has permitted a second approach to be defined for the implementation of this parallel logic system. By holding work locally and only transferring it when major imbalances were flagged, it was anticipated that the communication overheads would be substantially reduced. The question then became one of determining how successful a given strategy would be at maintaining a good distribution of work throughout the machine.

Information on the computational characteristics of the parallel declarative language system has been obtained from a pseudo-parallel version running on a single transputer implementation [24]. Different programs give rise to a wide range of values for these characteristics, and during one program run the individual process execution times can vary considerable as can the sizes of data packets representing results or spawned process data. Using this information as input to the application model generator, encoded trees can be produced to capture the details of program behaviour. In the second stage, the application executor "executes" the application from the data held in the encoded tree. In this case the evaluator models the logic interpreter in a manner that is indistinguishable from the fully functional one in its real-time behaviour and memory requirements. Finally, the raw data produced by the application is processed by the analyser tools and is presented to the user in graphical form.

As discussed earlier, load balancing strategies have long been in the focus of research and it seems that receiver initiated diffusion (RID) is the most favourable strategy for highly parallel, distributed memory systems. Under this general scheme a large variety of concrete RID algorithms can be realised, depending on what concrete design decisions are made.

In order to provide effective execution of this special class of decomposable applications, the processor farm mechanism is replaced with a *processor commune* in which each processing element interacts with its neighbours to reallocate work as necessary, and there is no global control of load balancing. In this system, each processing node (Figure 3.4) holds a copy of the evaluator (ie. the program in Figure 3.1) and the load balancing software components. This is known as SPMD (Single Program/Multiple Data) model. Figure 3.4 shows the relationship between the application software (evaluator) and the load balancing system (Balancer, Task store, Feeder). The software has been defined as a number of concurrent processes which coordinate their activity by message passing operations. The separation of these components allows different applications to be used with the load balancer. Work distribution is achieved by passing the input and output records (representing task data) between neighbouring nodes. The processor topology is fixed at the start of runtime.

The investigation has involved a detailed study of localised nearest neighbour load balancing strategies. Our model has to perform well in the situation where the total number of tasks and their individual processing times are unpredictable in advance. It has become clear in the course of the research that maximum performance benefit occurs when the primary objective is to minimise the load balancing overheads, rather than ensure an even spread of work throughout the machine.

Under the processor commune approach, newly decomposed or spawned tasks are stored locally to await execution unless the balancer process intervenes to export work because work loads are becoming unacceptably uneven. The balancer process is respon-

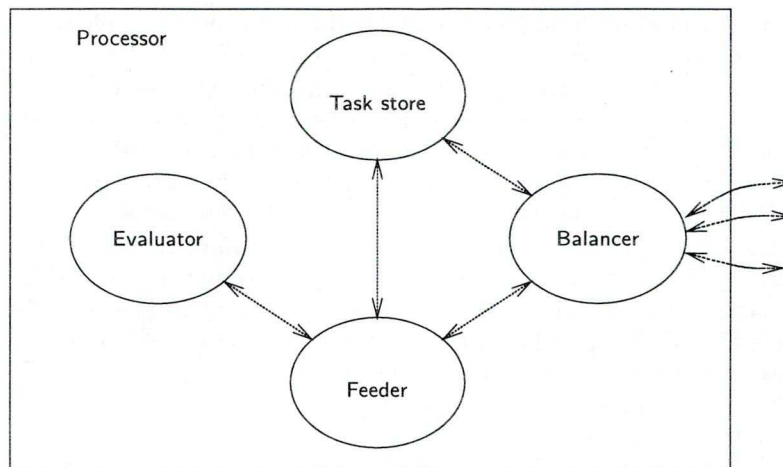


Figure 3.4: Application Executor Software Outline

sible for the monitoring of its work load, communications with neighbouring processing elements and relocation of data records representing tasks awaiting execution where appropriate. These tasks form the work load that is subject to dynamic load balancing.

Two factors are of importance in achieving effective decentralised load balancing in multi-processor networks: the frequency with which work loads are monitored and the overheads caused by the monitoring system itself. In order to prevent the load balancing inequalities throughout the machine, frequent monitoring is called for but this has to be weighed against the possible high overheads in terms of processor usage and communications costs. Our objective has been to establish a work allocation strategy which meets the following objectives:

- each processing element is kept busy throughout program execution
- load balancing overheads are very low

This has been achieved by the balancer applying an asynchronous approach to the monitoring of work loads. Each processing element works independently of its neighbours until the amount of work awaiting processing passes a certain threshold. Only then is the balancing system activated within that element. The balancing system is based on the weighting of work loads within each processing element, as defined by the four categories—EMPTY, LIGHT, AVERAGE and FULL.

The feeder process schedules work for the evaluator and reports to the balancer as work is performed. The balancer is responsible for initiating a session with neighbouring processing elements when the local work store (task store) holds too few or too many waiting tasks. On receipt of information on the load status of the neighbouring processing elements, the balancer may effect a work transfer.

The task weightings used in these investigations are defined in the following manner:

- | | |
|---------|--|
| EMPTY | — There are no tasks waiting evaluation |
| LIGHT | — The processor is likely to run out of work |
| FULL | — Local memory is full |
| AVERAGE | — All remaining cases |

While the task weighting remains *AVERAGE* within a processing element, no load balancing activity is initiated. However when this status changes, the “unbalanced” processing element prompts information exchange with its neighbours. The flow of tasks is dependent on the relative weightings. For each pair of balancing processors, if the weights are the same, no action is taken, otherwise the less loaded node receives work from the heavier one. The work transferred represents a group of sibling tasks, produced by one spawning operation, and stored as task initialisation data in a single data packet. Sibling tasks share common environment information, and thus the construction of a separate data packet for each individual spawned task would result in wasted storage space. The transfer of sibling tasks aids efficiency of execution by maintaining locality where possible. Only in the situation where a sender is itself has only one remaining data packet is the combined data packet split up and one individual task transferred. This means of representing a group of stored tasks not only conserves memory within the processing elements but minimises communication bandwidth requirements.

In the majority of cases, the load balancing initiation will occur because a *LIGHT* loading has occurred. The determination of an appropriate value of *LIGHT* is therefore of prime importance. A processing element is deemed to be in a *LIGHT* state if it holds a specified, small number of tasks waiting evaluation.

Thus in this system, monitoring of loads on neighbouring processing elements occurs only when the local work loads demand it. As execution proceeds, the feeder process passes work to the evaluator and receives any newly spawned tasks from it. The task store is responsible for all the storage management of the waiting tasks. If the number of these waiting tasks means that the work load weighting becomes other than *AVERAGE*, a session with neighbouring processing elements is established. Work is then transferred between processing elements if appropriate. To summarise, the components of the load balancing mechanism employed are as follows:

processor load evaluation: This is based on the definition of four values, *EMPTY*, *LIGHT*, *AVERAGE* and *FULL*. These are simple and quick to compute.

load balancing profitability determination: The determination of cost benefit to be made by transferring a task is made on the basis that

- ▶ the computation time of each task is relatively high, and
- ▶ the demand driven approach nature of the load balancing algorithm means that unnecessary communication is avoided. In this situation transfer of work between nearest neighbours is presumed to be efficient.

task migration strategy: The decision about whether work is to be transferred is taken in a bi-lateral manner between two processing elements, based on their relative weightings. This computation requires information exchange but is itself trivial. Hence in the situation where inter-processor communication does not delay processing, the operation is efficient.

task selection strategy: No application driven evaluation of which tasks to transfer has been made, other than the decision to allocate sibling tasks by one communication. Any group of spawned tasks is a candidate for relocation — the actual one selected is dependent on the task store and is chosen in the most efficient manner. However the separation of task storage and management from the work transfer mechanism means that it would be straightforward to install different task selection algorithms if required.

relationship among tasks: Hierarchical, as dictated by decomposable problems.

task granularity: Supposed to be relatively high, as is typical at the prime target area, the execution of declarative languages.

3.4 Tests

The generation of performance data and its subsequent analysis gives rise to a number of problems. Issues involving global versus local clocks may require attention, and intrusion occurs in performance monitoring which on occasions may affect the behavioural characteristics of the system [22]. In our system the overheads involved in time sampling and recording actions are sufficiently small, and for the purpose of this analysis can be neglected.

However the analysis of performance of parallel systems depends on the provision of a sound basis for comparison between the equivalent sequential and parallel versions [50]. It is fortunate that in our system it is possible to determine the best sequential time. This time is simply the sum of the execution times of the tree nodes. Obviously no sequential implementation can beat this limit. So we are going to use the "ideal" sequential execution times as our basis when presenting performance data and therefore the speedup figures obtained will be reliable, even a bit pessimistic.

Parallel programs do not tend to produce the same timing data every time they are run, the differences can be quite big. In our tests the discrepancies were larger with small-sized inputs and become very small with other inputs. We measured the execution times of the algorithm with the same tree in 16 successive runs and found that the absolute difference between the longest and the shortest run is less than 1% of the average running time for every but the smallest trees. The reason for this behaviour is the coarse granularity of the applications examined.

3.4.1 Test data

In order to test our system we generated a series of test trees with characteristics which we considered to be typical for the application area. Table 3.1 summarises the parameters given to the application model generator. Table 3.2 shows the most important properties of the trees, ie. the number of nodes they contain and the time they take to process assuming sequential execution without overhead.

	Spawning	Result	Fail
Ratio to total number of nodes	5%	10%	85%
Execution times	2- 16 msec	1- 10 msec	1-10 msec
Record sizes	32-126 bytes	16-126 bytes	N/A
Number of descendants	5- 25	N/A	N/A

Table 3.1: Test tree properties

Values were selected at random for individual nodes. Considering the speed of the transputer links we can say that these test "applications" are coarse grained, the time required for processing a task is at least an order of magnitude higher than sending it to a neighbouring processor.

number of nodes	total execution time (seconds)
1881	10.949
9551	55.305
13880	80.412
24960	143.913
32884	189.693
41534	239.983
49532	285.166
57053	329.967
65540	378.737
75465	435.693
84366	487.860

Table 3.2: Test tree data

3.4.2 Processor topology

Our algorithm does not depend on the physical connectivity of the processors, again giving an opportunity to experiment with different topologies and compare their effectiveness. In load balancing systems, where load redistribution takes place, obviously a small-diameter network is preferable [50]. In addition, as our system consists of uniform components, ie. all processing elements behave the same way, we speculate that the topology used should be homogeneous in the sense that the nodes cannot be distinguished from each other by examining their connection pattern. Other aspect to consider could be easy extendibility. The same topology has been used throughout the tests, but with varying numbers of processors. Figure 3.5 shows the modified ring with 12 processors.

3.4.3 Results

A number of tests were performed to obtain information on the performance of the load balancing system with different weighting parameters, tree sizes and processor networks.

The most indeterminate point in the workings of the load balancing algorithm is the definition of the LIGHT weight. How do we determine when a processing node is likely to run out of work? For these experiments, the number of waiting tasks is used as the measure of work load. We can set the LIGHT weight value to a fixed value (which may depend on the application and processor topology etc.) or we can apply some form of adaptive definition, allowing the threshold to change in time.

The question of determination of spawn ratio is of importance: for many systems it may not be possible to determine this in advance of program execution. However in these circumstances it is possible to compute the spawning ratio as it represents the ratio of the number of spawned to the total number of tasks. This computed value would not be precise at the start of run time, but would become more so as execution advanced.

The criterion for LIGHT loading is based on the work awaiting execution. In the initial series of tests, experiments were carried out under two different LIGHT definitions. Under the first approach a processor was deemed to be in a LIGHT state if it held at least one data packet, and the number of waiting tasks was less than the computed spawn ratio (Figure 3.6). It is important to note that the terms 'data packet' and 'waiting tasks' are not synonymous. One data packet can contain several waiting tasks: sibling tasks are stored in one packet, because they share common environment information and setting up a separate

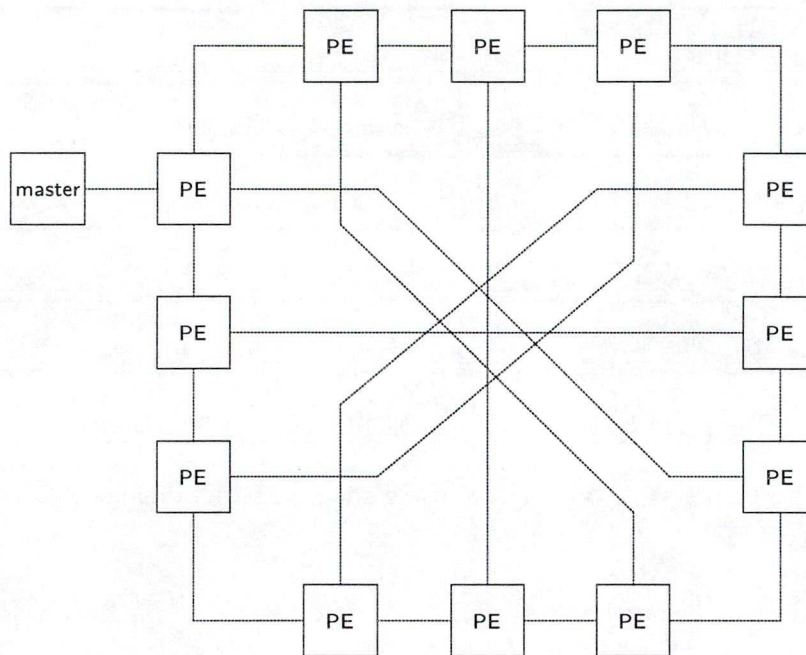


Figure 3.5: A 12 node processor topology

data packet for each individual spawned task would result in wasted storage space. In the second variation no account was taken of the number of data packets, a processor's loading became LIGHT when the total number of waiting tasks fell below four (Figure 3.7).

The diagrams show the work load on each transputer (represented by the number of waiting tasks) as a function of time. Vertical lines in the narrow band below each graph indicate the points in time when that particular transputer has run out of tasks waiting execution.

There is a strong relation between the shape of the diagram and the efficiency of the system. Consider the two primary requirements which a load balancing method must meet: first, the balancer has to ensure that there is always be at least one process waiting for the evaluator to execute. This is likely to involve communication with the balancer processes on neighbouring transputers. Secondly, as the balancer and the evaluator run on the same processor, the time involved in the balancer's activities should be minimised in order that the operation of the evaluator is not unduly penalised. This is important because it is the evaluator who does the "real" job. Unfortunately these two requirements conflict.

When the output of the load balancing analyser is studied, a number of interesting features can be identified. Figure 3.8 shows an enlarged section of part of a typical workload graph. A detailed examination of this reveals that in the sections marked 'a', the evaluator proceeds without intervention of the balancer. The waiting tasks are processed, one after the other. However at times 'X' and 'Y', the number of waiting nodes increases suddenly. This could be either due to the occurrence of a spawning operation or to balancer intervention. In the latter case the balancer is the recipient of an incoming data packet, either because the local processor is running short of work, or because it has accepted one from an "overflowing" neighbouring transputer. The reverse operation can be seen at time 'Z': here

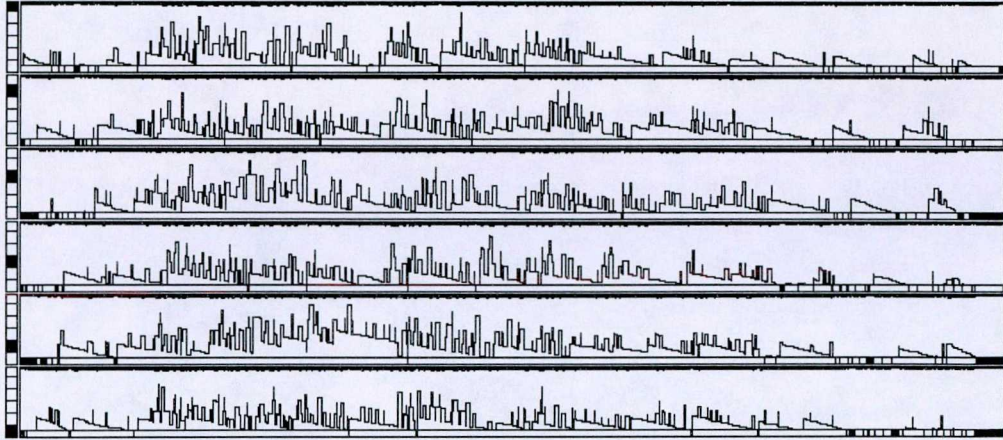


Figure 3.6: Processor load using adaptive LIGHT definition

the sudden drop in the number of waiting tasks indicates that the balancer has transferred one or more packages to a neighbour or neighbours.

The two graphs shown in Figure 3.6 and 3.7 differ considerably in the degree of irregularity they exhibit — the first one contains many of these “thorns”. Such frequent transfers are unnecessary and mean that overall performance is likely to be worse than in the second case because of the processing overheads involved. For both systems the number of occasions on which a transputer is idle because it has run out of waiting processes is small, and thus both balancing strategies ensure that little processor time is wasted. It is worth noting that the first approach is actually achieving a more even spread of work throughout the machine, albeit at the expense of frequent work transfer. Based on these observations, small LIGHT constants are preferable for our applications.

It is clear that the effects of varying the details of load balancing strategies that the factors involved in the evaluation of load balancing strategies are necessarily complex, and the test system provides a good environment for their exploration.

Figure 3.9 shows the speedups acquired by setting the LIGHT threshold to 3 and using different numbers of processors. The good speedup figures demonstrate that the load balancing system is working efficiently and not imposing heavy overheads on the system, it performs well with this type of coarse granularity applications. The LIGHT weighting is sufficient to prevent processing nodes running out of work while work exchange is taken place, but not so high as to invoke frequent (and possibly unnecessary) transfers.

Future plans include the use of the test environment to fully explore how the different parameters influence the effectiveness of load balancing.

3.5 Application evaluators

3.5.1 Application mimic

The application executor was first implemented in parallel C, later in OCCAM running on transputer networks and has two components, the evaluator and the balancer. The default evaluator in the test system “executes” the application model by running idle loops in place

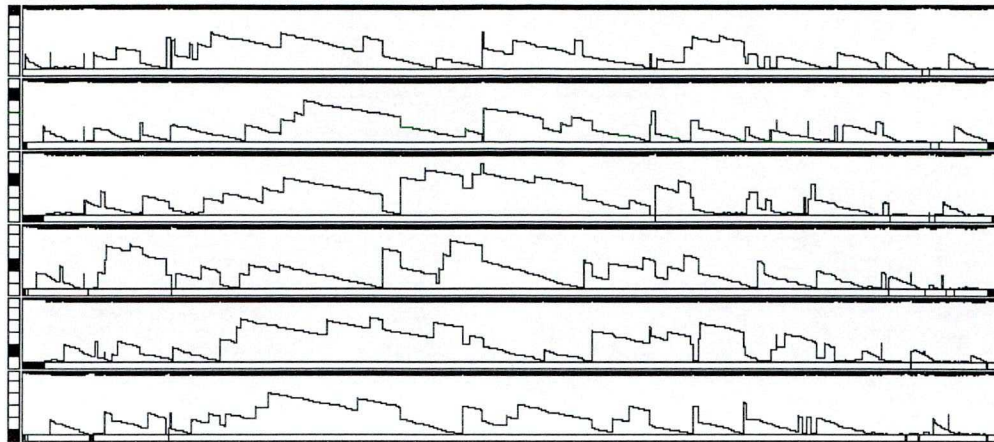


Figure 3.7: Processor load with LIGHT threshold = 3

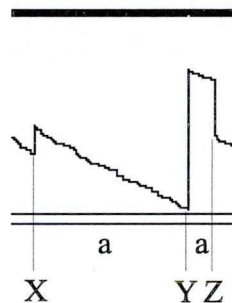


Figure 3.8: Enlarged section of processor loading graph

of application tasks, producing empty data packets of sizes as prescribed. It can be, however, easily replaced by a "real" evaluator which meets the simple interface requirements to the balancer.

3.5.2 PROLOG interpreter

A real evaluator, a fully functional pure PROLOG interpreter has been implemented by a graduating student under supervision and performance measurements show that work is indeed distributed evenly among processors [58]. It is worth mentioning that full PROLOG [32] has a number of features that are inherently sequential as they need global information about the program state. These features make a full parallel implementation hard to realise. This theme brings us again to programming models and in fact experience with the PROLOG system heavily influenced the definition of the NOP model.

3.5.3 Attribute grammar evaluation

Another area of investigation was that of an attribute grammar based compiler generator system where attribute evaluation takes place in parallel [6]. Compiler generator systems

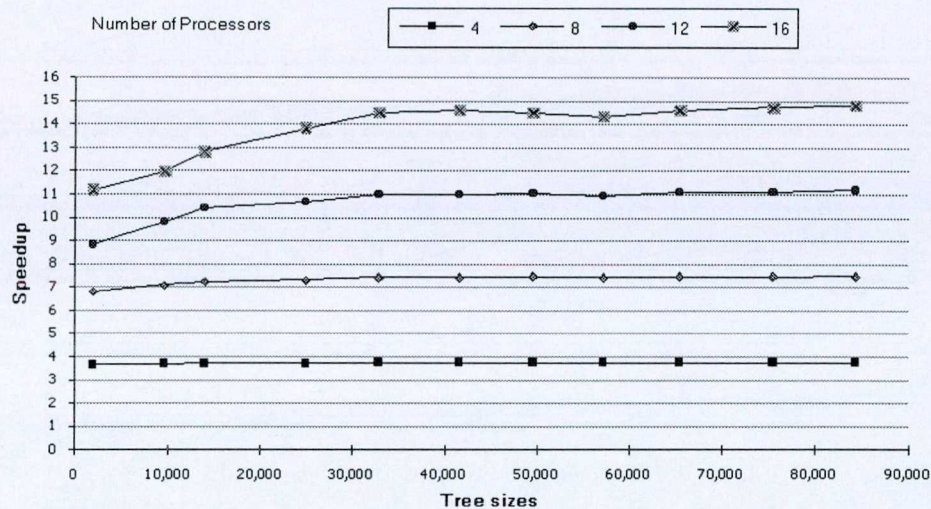


Figure 3.9: Speedup figures

generate executable compilers from formal specifications, most of recent compiler writing systems are based on attribute grammars. Experiences with compiler construction proved the feasibility and efficiency of attribute grammars for compiler specification.

Several parallel attribute evaluation methods are known, they exploit the fact that independent attribute instances can be evaluated in any order [57], even in parallel. The problem with fully automatic methods is that they cannot distribute the work among processing elements satisfactorily because the time to evaluate the semantic equations is unpredictable and can range from very short to extremely long. To handle the problem, a semi-automatic evaluation strategy was proposed in which the user (the author of a particular grammar) annotates the source code to help the system determine which attribute instances are good candidates for parallel evaluation. With this strategy, a dynamic load balancing evaluator can be expected to work with greater efficiency. Future plans include the application of the test system to carry out a series of test which determine the boundaries of the effectiveness of the commune load balancing model.

3.6 Implementation

By the end of the development of the parallel C version of the test system, based on the experience gained through the tests, it became apparent that OCCAM provides much better facilities for writing concurrent applications [47][19]. Among its chief advantages are the seamless integration of parallel and communication primitives into the language, the possibility of declaring channel protocols, the strict type checks on data communicated on channels. Besides these, OCCAM's constructs are well defined, making it easier to discuss program correctness. All these factors influenced the decision that the system should be rewritten in OCCAM, the result of which is presented in Chapter 5 after gaining some insight into the workings of the commune load balancing algorithm.

Chapter 4

Correctness of the load balancing protocol

Based on [8].

4.1 Introduction

Except for the most trivial cases, the correct behaviour of programs is far from obvious and rapidly becomes less and less so as complexity increases. Parallel systems are typically amongst those of great complexity, so our confidence in their proper behaviour should be strengthened by some form of validation. Validation can be carried out after the system has been designed (and possibly implemented) by enumerating and checking the system states. However, as the number of system states grows exponentially when the number of system components or number of component states grows, this form of validation is limited [65].

Another possibility is to ensure correctness in the design phase. One advantage of this method is that we can use building blocks (components whose correctness have been proved) to construct larger systems. Once the correct behaviour of the building blocks is ensured, the system composed of these blocks is also correct as far as the interface requirements of the blocks are observed. A good example in this category is the client-server type systems [66]. In this approach processes are constrained to act in active (client) or passive (server) modes with respect to communication interaction. By following client-server interaction rules deadlock/livelock freedom can be guaranteed.

Synthesized from work with the processor commune system, a building block has been defined which enables processes to run independently and communicate with each other without a predefined communication pattern. This means that at any given time it is not known in advance which of two processes will send a message to the other. The processes work independently as long as they can, connection is established only when needed. The basic building block is successively extended in order to be able to deal with more complex situations.

The setup for the problem consists of two processes connected by a pair of channels, they want to communicate with each other at unpredictable moments without risking getting deadlocked. All protocols were defined in OCCAM.

The first (*basic dynamic interaction*) version of the building block enables the two parties

to exchange simple messages without deadlock. The second (*bounded resource dynamic interaction*) version introduces a *store* process which stores items to be exchanged between the two parties. There are a limited number of items, this restriction complicates the protocol, but deadlock freedom can be guaranteed.

The third version (*general dynamic interaction*) merges the former two. It stipulates that components use two types of data during their activity: *control data* and *mass data*. Control data items are small and carry control information, they do not accumulate but are processed upon receipt. Mass data items are large and carry the workload to be processed by the system, they accumulate if arrive faster than processing takes place. In fact the basic and bounded resource versions deal with these two types of data respectively. It is possible to combine them in a way that guarantees correct functioning. In general, correctness means that all specification requirements are met by the system. While most requirements can be problem specific, one of them is universal: absence of deadlock and livelock, and this is what is meant by 'correct' in this general setting. Beside this, the issue of fair resource handling is addressed where appropriate.

A building block suitable for using in parallel systems contains a coordinator *boss* process, a *store* process, and any number of general dynamic interaction *worker* processes. Such a system of processes is deadlock free.

In the load balancing test system, each processing element has a copy of the general building block described above and as such, the load balancing system is deadlock free. In this particular case, a circulating token is used to detect termination of the computation [67], the token is implemented as a control data item. The tasks to be processed are of course mass data items.

4.2 Problem description

Our aim is to define a building block which enables processes to run independently and communicate with each other without a predefined communication pattern. This means that at any given time it is not known in advance which of two processes will send a message to the other. The processes work independently as long as they can, connection is established only when needed. This situation is typical of several classes of systems, eg. dynamic load balancing kernels, reactive, real time systems, distributed object systems, operating systems. The basic building block is successively extended in order to be able to deal with more complex situations. The dynamic process interaction scheme has been applied to implement the processor commune load balancing system described in the previous chapter. Throughout the paper we use the OCCAM model of parallelism [45] as a protocol definition language. OCCAM is based on CSP [39][40] and therefore has a firm theoretical background, making it suitable for this task.

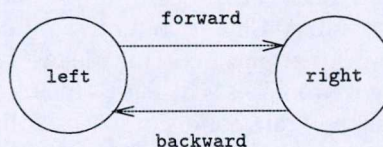


Figure 4.1: Basic setting

Parallelism helps us in speeding up a computational task by using several computational units, each working on independent parts of the same task. In the OCCAM model,

an abstract computational unit is called *process*. Consider the system in Figure 4.1 consisting of two processes (*left*, *right*) working on parts of a task in parallel and connected by a pair of channels (*forward*, *backward*). The two processes are assumed to be of equal “priority”, meaning that none of them is a subordinate to the other. With this assumption we ruled out client-server type systems at this level. Quite probably the task to be solved cannot initially be split up into two parts which require equal processing time, or the subtasks are not completely independent. In either case one of the processes may run out of work and cannot continue. As an idle process does not help in speeding up the computation, it should be provided with the data it needs to be able to continue working. For the data transfer, the processes have to establish a connection.

The problem is that in general the time and direction of communication is unpredictable in advance as it depends not only on the algorithm applied, but also on the particular task to be solved. Theoretically, we can send messages at any time, but this ad hoc method is very dangerous as it can cause deadlock in the system. To avoid deadlock, a predefined communication pattern of monitoring messages can be applied, but this method incurs unnecessary communication overheads. Our aim is to let the processes work undisturbed while they can and to provide a safe connection establishing mechanism to allow the processes communicate if the need arises.

4.3 Dynamic interaction skeleton

These above observations lead us to the following problem formulation.

4.3.1 High-level structure

Consider the system in Figure 4.1. Suppose the processes want to send messages to each other at moments that are unpredictable in advance. For the time being, the contents of the messages are not important, the only requirement is that deadlock does not occur. As the role of the component processes is completely symmetrical, they will be controlled by the same program code. List 4.1 shows the high-level description of the system using pseudo-OCCAM.

```

1 PROC component(·CHAN·OF·TRANSACTION·inp,·out)
2 ······
3 ······code of component
4 ······
5 :
6 CHAN·OF·TRANSACTION·forward,·backward:
7 PAR
8 ··· component(·forward,·backward)·····process·right
9 ··· component(·backward,·forward)·····process·left

```

List 4.1: High level description

In order to be able to use more human terms in the discussion, let us use an analogy. Suppose the component we want to define is a small firm. It has only two members, one *worker* and (of course) one *boss*. These agents act in parallel as can be seen in Figure 4.2.

4.3.2 Structure of the component — the firm

The channels from the outside world are connected to the *worker*. There is a local channel on which the *boss* sends its requests to the *worker*. The duty of the *worker* is to faithfully

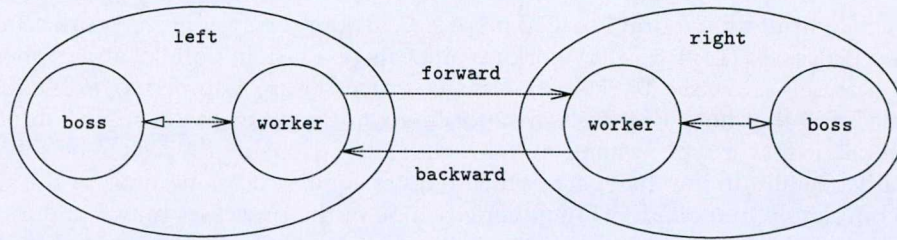


Figure 4.2: Two firms

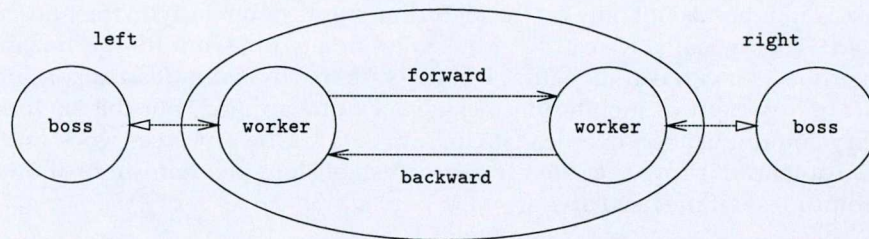


Figure 4.3: Two firms — alternative view

carry out the requests and report back the outcome. There is another channel for the answer pointing from the *worker* to the *boss*. This pair of channels is used in a client-server fashion, the *boss* being the client. This is emphasized by a triangle-shaped arrowhead. List 4.2 shows the *firm* process described in pseudo-OCCAM. The rest of this chapter is going to be devoted mainly to define the *worker* process and to extend the *firm* with other components so that it could perform more complex tasks.

4.3.3 The skeleton

Figure 4.2 shows the structure of a two-firm system as it is implemented. It must be noted however that in subsequent discussions we regard the *boss* processes as the environment of the pair of *workers* and examine the correctness of the system in this context (Figure 4.3). Deadlock/livelock freedom means that the pair of *workers* are always able to communicate on the request channels. The *worker* has to accept messages both on its request and input channels. If the message has come from the *boss*, it should forward it on its output channel and acknowledge. If the message has arrived on the input channel, it should accept it, but no further action is required.

The first “solution” that comes into mind can lead very easily to deadlock (List 4.3). The problem is that deadlock occurs if both sides decide to send a message to the other because both of them are obliged to wait at their respective `out ! request` statements (line 37) for the other side to accept the message. But the other side is also waiting, so no further progress is possible. There is also a more subtle point that can cause problems: a *boss* on the left side can be indefinitely intercepted if the *boss* on the right side keeps sending messages and these are handled in the passive phase on the left side. This is possible because the `ALT` statement is nondeterministic by definition.

The problem is solved by adopting the following modifications (List 4.4):


```

10 PROC firm(·CHAN·OF·TRANSACTION·inp,·out)·---
11 ··CHAN·OF·REQUEST·req:
12 ··CHAN·OF·BOOL·...·ans:
13 ··PROC·boss()·-----
14 ····REQUEST·request:
15 ····BOOL·...·answer:
16 ····WHILE·TRUE
17 ······SEQ
18 ······--·...
19 ······request·:=·decision.making()
20 ······req·!=·request
21 ······ans·?.·answer
22 ······--·...
23 ··:·--·boss·-----
24 ··PROC·worker()·-----
25 ····--·code·of·worker
26 ··:·--·worker·-----
27 ··PAR
28 ····boss()
29 ····worker()
30 :·--·firm·-----

```

List 4.2: Firm process

```

31 PROC worker()·---·deadlock·version·-
32 ··REQUEST·request:
33 ··WHILE·TRUE
34 ····ALT
35 ······req·?.·request·---·active·phase
36 ······SEQ
37 ········out·!=·request
38 ········ans·!=·TRUE
39 ······inp·?.·request·---·passive·phase
40 ······SKIP
41 :·--·worker·-----

```

List 4.3: Deadlocking worker process

- ▶ Every communication between the two sides will consist of two messages, one in *each* direction.
- ▶ The *active phase* is coded using a PAR construct (48) to avoid deadlock when the other side is also trying to send a message.
- ▶ The *passive phase* is coded so that it sends back an acknowledgement message (55), thus allowing the termination of the active phase on the other side.
- ▶ PRI ALT is used (45) instead of ALT to ensure that the boss is selected when ready thus avoiding the starvation effect.

We would like to be certain that neither boss is locked out indefinitely from communicating with its worker. Let us examine the workings of the system with the above worker definition (List 4.4). We will see that in any event the worker gets back to the beginning of the WHILE loop, ie. to the PRI ALT statement (45). This is exactly what we need.

When the system is started, both workers wait on their PRI ALT statement, the *initial state*. At this point two things can happen from the point of view of the left side:

- ▶ The boss sends a message on the request channel (active phase at line 47). Now the right side is:
 - either in its active phase: messages on the two connecting channels go in parallel and on both sides the active phase terminates. The workers return to their initial state after sending an acknowledgement (51) to the bosses.


```

42 PROC worker() -- skeleton version --
43 .. REQUEST reqInp, reqOut:
44 .. WHILE TRUE
45 .... PRI ALT
46 ..... req ? reqOut -- active phase
47 ..... SEQ
48 ..... PAR
49 ..... out ! reqOut
50 ..... inp ? reqInp
51 ..... ans ! TRUE
52 ..... inp ? reqInp -- passive phase
53 ..... SEQ
54 ..... reqOut := reqInp
55 ..... out ! reqOut
56 :- -- worker: -----

```

List 4.4: Skeleton worker process

- or waiting at PRI ALT: the out ! reqOut (49) statement on the left side puts the right side into passive phase. The passive side sends back an acknowledgement (55), this terminates the PAR (48) on the left side. Again both sides go back to the initial state.
- A message arrives on the input channel (passive phase). This is the same situation as the one just before, only the roles of the left and right sides are reversed.

So we can conclude that the system works correctly. We are now in a position to extend our system to allow it to deal with more complex situations. We note however that the interaction between the two sides is going to follow the pattern defined above (active, passive phases). This means that the proof of correctness will require the same considerations, ie. we have to check how the system works when both sides are in active phase (active-active case), and when one side is in active phase, the other is in passive phase (active-passive case).

4.4 Basic dynamic interaction — BDI

The first extension to our skeleton system is an extension to the vocabulary of messages. In the preceding part the messages could be anything, only their presence mattered, not their content. Now we stipulate that the worker of a firm stores its state information in a single variable. The purpose of this extension is to let the firms transfer the values of these state variables to each other in message *items*. Now the boss can issue requests of two kinds, *send* or *receive*. On receipt of a *send* message from its boss, a worker is obliged to send an item to the neighbour. On receipt of a *receive* message, the worker has to acquire an item from the neighbour. In both cases an acknowledgement is sent back to the boss after transfer. If *send* or *receive* arrives from outside, the worker has to act accordingly, ie. accepting an item if a *send* message arrives and send one if a *receive* message arrives. It is again possible that the two sides initiate a transfer at the same time. If the messages are the same, a pair of items are exchanged, if they are different, a single item is transferred in the appropriate direction.

The processes (List 4.5) use a tagged PROTOCOL (57) for this type of interaction, differentiating between an *initialisation phase* (59) and a *transfer phase* (60). The list shows the modified code for the worker.

The transfer (72, 78) procedure is relatively straightforward so its definition is omitted for now (see next Chapter). It receives two parameters indicating the directions as


```

57 PROCEDURE TRANSACTION. -----
58 .. CASE
59 .... init; REQUEST. -- 'send' or 'recv'
60 .... xfer; ITEM
61 .. TRANSACTION. -----
62
63 PROC worker(). -- basic version. -----
64 .. REQUEST reqInp, reqOut:
65 .. WHILE TRUE
66 .... PRI. ALT
67 .... req ? reqOut. -- active phase
68 .... SEQ
69 .... PAR
70 .... out ! init; reqOut
71 .... inp ? init; reqInp
72 .... transfer( reqInp, reqOut)
73 .... ans ! TRUE
74 .... inp ? init; reqInp. -- passive phase
75 .... SEQ
76 .... reqOut := invertReq( reqInp)
77 .... out ! init; reqOut
78 .... transfer( reqInp, reqOut)
79 .. worker. -----

```

List 4.5: Worker — Basic version

requested by the neighbouring and the local boss respectively. It communicates with the transfer procedure on the other side.

In the active-active case, the directions are exchanged ([69]), then the transfer procedures transfer the item(s) as appropriate. In the active-passive case, the passive side sends back the opposite of the message received (*invertReq* function [76]), so a single item will be transferred in the direction requested by the active side. This is also why the value *TRUE* is sent back as an acknowledgement to the boss ([73]). We can see that the system works correctly.

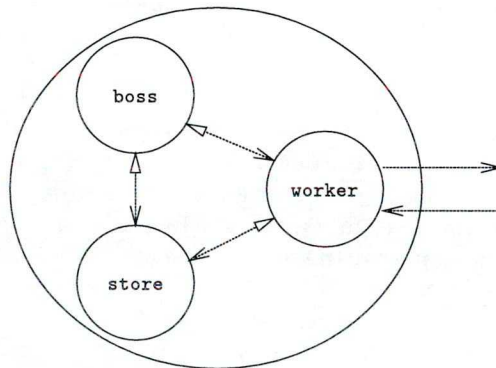


Figure 4.4: Firm with store

4.5 Bounded-resource dynamic interaction — BRDI

We now examine the case when a limited number of items can be stored by each firm. These items are stored by a new member: the storekeeper, whose duty is to serve the other two who act as clients (Figure 4.4). We could write the code for the communication

with the store, but that is unnecessary for our present purposes. What is important is the definition of the interface through which these services are accessible. It should be possible to get items from and put items into the store if it is not empty or full respectively. The following procedures will be used for these operations. They involve communication on the channels connecting to the store.

- PROC book(REQUEST req, BOOL booked):
 - if called with a *send* first parameter, it checks whether there is an item in the store which can be removed. If so, booked will be TRUE and an item will be reserved for a subsequent getitem operation. If the store is empty, booked will be FALSE.
 - if called with a *receive* first parameter, it checks whether there is room in the store for another item. If so, success will be TRUE and an empty slot will be reserved for a subsequent putitem operation. If the store is full, booked will be FALSE.
- PROC cancel(): cancels the previous booking. Every client can have only one outstanding order, so a parameterless cancel is sufficient.
- PROC getitem(ITEM item): removes a previously booked item from the store.
- PROC putitem(ITEM item): places the item into a previously booked slot in the store.

The worker process needs alteration to deal with the new situation (List 4.6). Now if a request arrives from the boss, it first has to check the state of the store whether the request makes sense at all (87). Similarly, if a request arrives from outside, the store has to be checked to decide whether or not to initiate a transfer (111). Further, if in the active-active case the same message is sent by the two sides, then no transfer will take place. This is because the items are considered to represent the runtime data under processing. When firms exchange two items, their load does not change, so it is not worth the trouble. There may be applications where this behaviour is not desirable, it easy to modify the procedure to force the exchange of items.

Again, we should examine whether the system works correctly. There is no danger of starvation so let us concentrate on deadlock. Let us consider the case when both sides are active. If book fails (101,106), after reporting the failure (108) the worker returns to the initial state, so we have to consider only the case when book succeeds. If it does (90), both sides check whether the messages are the same. If they are, a cancel operation is performed (97) as book was successful yet we do not initiate a transfer because the direction is not agreed on. The boss is notified of the outcome by sending the value of the variable booked to it (108). We can conclude that in the case when both sides are active the system is deadlock-free and the item store is handled according to the usage rules.

In the passive phase, if book is successful, the behaviour is the same as with the basic version, so we have to deal with the case when book fails. If it fails, the received message is sent back (121) and the process returns to its initial state. The process on the other side successfully finishes its PAR, compares the outgoing and incoming messages (95), finds that they are the same, cancels the booking and notifies the boss. After this, it also returns to the initial state. Again, if we take into account that in the passive phase transfer is invoked only when the booking was successful, we conclude that the active-passive case is free from deadlock and resource handling is correct.

From the above discussion we can see that the system works correctly with limited resources.


```

80 PROC worker(). --- limited storage version --
81 .. REQUEST reqInp, reqOut:
82 .. BOOL ... booked:
83 .. WHILE TRUE
84 ... PRI ALT
85 ... req ? reqOut --- active phase
86 ... SEQ
87 ... book( reqOut, booked)
88 ... IF
89 ... booked
90 ... SEQ
91 ... PAR
92 ... out ! init; reqOut
93 ... inp ? init; reqInp
94 ... IF
95 ... reqInp = reqOut
96 ... SEQ
97 ... cancel()
98 ... booked := FALSE
99 ... TRUE
100 ... SKIP
101 ... TRUE
102 ... SKIP
103 ... IF
104 ... booked
105 ... transfer( reqInp, reqOut)
106 ... TRUE
107 ... SKIP
108 ... ans ! booked
109 ... inp ? init; reqInp --- passive phase
110 ... SEQ
111 ... book( invertReq( reqInp), booked)
112 ... IF
113 ... booked
114 ... SEQ
115 ... reqOut := invertReq( reqInp)
116 ... out ! init; reqOut
117 ... transfer( reqInp, reqOut)
118 ... TRUE
119 ... SEQ
120 ... reqOut := reqInp
121 ... out ! init; reqOut
122 : --- worker -----

```

List 4.6: Worker — Limited storage version

4.6 General dynamic interaction — GDI

Now the BDI and BRDI schemes will be combined to yield a more powerful building block. It is assumed that firms use two types of data during their activity. The first type of data is called *control data* and is assumed to be stored in a single variable. Firms may ask for the control data of their neighbour or may initiate their own control data to be sent to the neighbour. In either case, the data received are either discarded or overwrite the former content of the control variable, ie. control data are not accumulative. Notice that BDI can be used to achieve exactly this kind of behaviour, the items used there are the control data values just described. The other type of data is called *mass data* and this is exactly the kind of data that items represent in BRDI. These items are supposed to carry the runtime data subject to processing.

Both types of data are held in the store. In order to be able to handle the resources uniformly, a modified booking procedure is defined, where we have to specify the type of data to be booked.

► PROC book(REQUEST req, REQTYPE type, BOOL booked)

Control data is always available, so book always succeeds if type is *ctrl*. The transaction protocol has to be modified to allow differentiating between the two types of data (123).

Besides the direction, each initialisation message has to contain the type of the item it proposes to transfer. Lists 4.7 and 4.8 show the code of the worker process for GDI. The difficulty in constructing this latest procedure is to ensure sensible and correct behaviour when the two sides try to initiate transfers of different types. For example, when one side tries to get an item of mass type data and the other side tries to get control data.

```

123 PROTOCOL TRANSACTION -----
124 .. CASE
125 ... init; REQUEST; REQTYPE -- 'mass' or 'ctrl'
126 ... xfer; ITEM --- same representation for both types
127 : --- TRANSACTION -----
128
129 PROC worker() --- general version -----
130 .. REQUEST reqInp, reqOut:
131 .. REQTYPE typeInp, typeOut:
132 .. BOOL ... booked, xferflag:
133 .. PROC corrector() ...
134 .. WHILE TRUE
135 ... PRI ALT
136 ... req ? reqOut; typeOut --- active phase
137 ... SEQ
138 ... book(reqOut, typeOut, booked)
139 ... xferflag := booked
140 ... IF
141 ... booked
142 ... SEQ
143 ... PAR
144 ... out ! init; reqOut; typeOut
145 ... inp ? init; reqInp; typeInp
146 ... corrector()
147 ... TRUE
148 ... SKIP
149 ... IF
150 ... xferflag
151 ... transfer(invertReq(reqInp), reqOut, typeInp, typeOut)
152 ... TRUE
153 ... SKIP
154 ... ans ! booked
155 ... inp ? init; reqInp; typeInp --- passive phase
156 ... SEQ
157 ... book(invertReq(reqInp), typeInp, booked)
158 ... IF
159 ... booked
160 ... SEQ
161 ... reqOut, typeOut := invertReq(reqInp), typeInp
162 ... out ! init; reqOut; typeOut
163 ... transfer(invertReq(reqInp), reqOut, typeInp, typeOut)
164 ... TRUE
165 ... SEQ
166 ... reqOut, typeOut := reqInp, typeInp
167 ... out ! init; reqOut; typeOut
168 : --- worker -----

```

List 4.7: General worker

Now let us consider whether the GDI works correctly. First let us assume that both sides are in active phase. Notice that if the data types the two sides try to transfer are the same then we get the same behaviour as from BDI (control data) and BRDI (mass data) respectively, so the correctness of the algorithm in these cases follows from that of BDI and BRDI. If the data types to be transferred are different, then there are four possibilities:

mass send $\iff$ ctrl send
 mass send $\iff$ ctrl receive
 mass receive $\iff$ ctrl send
 mass receive $\iff$ ctrl receive

An extra message is exchanged between the two sides to decide whether the mass item can be transferred ([181], [189]). In either case the values of the variables are modified in the corrector procedure to reflect the changes. Transfer is initiated only when necessary and

the boss gets back an acknowledgement reflecting the fate of its request.

When considering the active-passive case, it can easily be seen that the passive side responds as dictated by the active side. The correct behaviour thus follows from the fact that BDI and BRDI were correct. To summarise, we can say that GDI is correct, ie. deadlock/livelock does not occur and it handles both types of data according to the usage rules.

```

169 PROC corrector()· -----
170 ·· IF
171 ··· (· reqInp· =· reqOut)· AND· (· typeInp· =· mass)· AND· (· typeOut· =· mass)
172 ···· SEQ
173 ····· cancel()
174 ····· booked,· xferflag· :=· FALSE,· FALSE
175 ··· (· typeInp· =· mass)· AND· (· typeOut· =· ctrl)
176 ····· BOOL· success:
177 ····· SEQ
178 ····· book(· invertReq(· reqInp),· typeInp,· success)
179 ····· IF
180 ······· success
181 ······· out· !=· init;· invertReq(· reqInp);· typeInp
182 ······· TRUE
183 ······· SEQ
184 ······· out· !=· init;· reqInp;· typeInp
185 ······· reqInp,· typeInp· :=· invertReq(· reqOut),· typeOut
186 ··· (· typeInp· =· ctrl)· AND· (· typeOut· =· mass)
187 ····· REQUEST· request:
188 ····· SEQ
189 ····· inp· ?· init;· request;· mass
190 ····· IF
191 ······· request· =· reqOut
192 ······· SEQ
193 ······· cancel()
194 ······· booked· :=· FALSE
195 ······· reqOut,· typeOut· :=· invertReq(· reqInp),· typeInp
196 ······· TRUE
197 ······· SKIP
198 ····· TRUE
199 ····· SKIP
200 ·· --- corrector· -----

```

List 4.8: Corrector procedure

4.7 A building block for larger systems

Notice that because the store acts as a server, we can allow other processes to manipulate with it, as far as they act as clients. This additional permission does not affect the correct behaviour of the system (book() now takes into account the already booked slots). If so, let us employ some more workers and connect the boss to the store. The resulting *firm building block* is shown in Figure 4.5.

Notice further that a set of such blocks connected to each other arbitrarily forms a process that possesses a valuable property, namely that to the set of boss processes as environment, the rest of the system is deadlock free. This can be seen as follows: We know [66] that a set of processes communicating according to the rules of the client-server paradigm is deadlock free if there is no directed cycle in the client-server dependency graph. We can see that the building block is in fact complies with this requirement because

- ▶ every pair of workers is a process that acts as a client to two stores and acts as a server to two bosses.
- ▶ by substituting a process in place of every connected pair of workers we get an acyclic client-server graph.

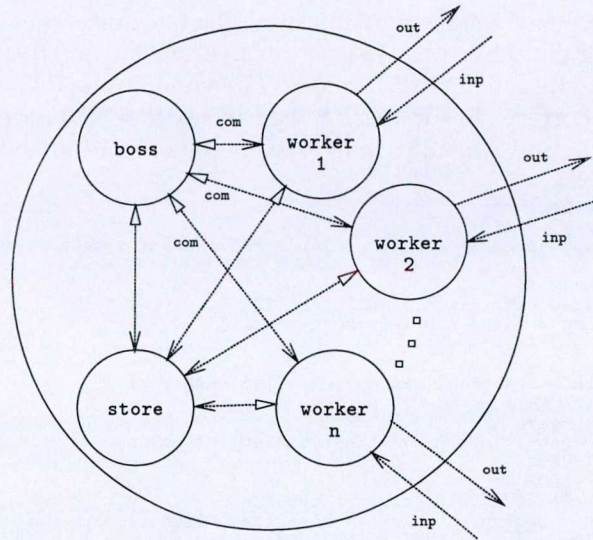


Figure 4.5: Firm with many workers

4.8 Applying the building block

The firm building block was used to build the processor commune load balancing system. Recall that a commune component consists of four processes: *evaluator*, *feeder*, *store* and *balancer*. These correspond to the processes of the firm building block in a natural way: *boss* stands for the duo of *evaluator* and *feeder*, *store* is evident, and *balancer* stands for the set of *workers*. Apologies to all those who find this proliferation of seemingly unrelated names annoying, hopefully the descriptions shed some light on the author's intentions.

The tasks to be processed are the mass data items in the store. For termination detection, a circulating token is applied. It is worth noting that the termination detection algorithm was also derived by successive refinement [67][68]. A control data variable was used to implement the token.

Chapter 5

Implementation of the load balancing system

5.1 Configuration

The source code listing can be found in Appendix B. Source lines will be identified with their sequence number appearing in the listing (eg. 123).

The source files are stored in several directories, each containing a logical unit. The following arrangement is used assuming a suitable base directory:

makefile	1	profile/		app/	
commune.lnk	126	profdiff.occ	1074	appifac.inc	1928
		profile.occ	1109	app.occ	1944
config/		profserv.inc	1235		
commune.inc	142	profserv.occ	1242	boss/	
commune.pgm	155			cmmnboss.inc	2028
		store/		cmmnboss.occ	2034
cmmnroot.occ	289	storifac.inc	1808	feeder.occ	2082
cmmnpars.occ	536	cmmnstor.inc	1375		
commune.occ	803	cmmnstor.occ	1445		
cmmnmain.occ	864				
sentinel.occ	2335				
guard.occ	2376				

The makefile contains the rules and commands used to generate the executable from the source files. The standard INMOS toolset [46] components are assumed to be accessible. The OCCAM source files are compiled into object files (all generated files are placed into the objT8/ directory). There are two processes on the highest level, one for the master processor linked in cmmnroot.lku (46) and another to be placed on the commune processing elements, linked in commune.lku (51) using the list of object files in commune.lnk (see also Figure 3.5).

The interface of the `cmmnroot` process (□295) contains two pairs of channels and an integer representing the number of commune nodes in the system. One pair of channels connect to the File Server on the host machine, the other pair is used to communicate with the commune components. The `commune` process interface (□805) contains its node *id*, two arrays of input and output channels to communicate with neighbouring commune members, another pair of channels to communicate with the environment and a block of memory for runtime data. The process network must be constructed in such a way that a Hamilton cycle can be formed. This cycle will be used to distribute initialisation data and collect data after processing has finished. Each `commune` process has at least two pairs of channels connecting it to its two neighbours in the Hamilton cycle. The one next to `cmmnroot` is called the *head* and has *id* = 0. The environment channels are used only by this process and are connected to `cmmnroot`. The Hamilton cycle is assumed to be directed, so each process has a pair of channels on the *backward* side and another pair on the *forward* side.

The configuration file `config/commune.prg` (□155) contains three sections, these are the processor network description, the software network description and the mapping section. Any processor network can be used which is compatible with the requirements above. One `commune` process should be placed on every processor, therefore the hardware and software networks have the same structure and links and channels correspond to each other fairly naturally.

The file `config/commune.inc` contains a few global constants. `HamBwSide` (□147) and `HamFwSide` are indexes used to select the corresponding channels from the array of input and output channels of a process. For example, `HamFwSide` is 0, so data on the first element of the array of input channels come from the next process on the cycle. `MaxNofNeighbours` is set to the maximum number of neighbours each process can have, this is limited by the number of links on the transputer.

The first segment of the configuration declares constants to be used to build the modified ring of processors in Figure 3.5. The links to be connected must be carefully selected because of the architecture of the T.Node machine [20].

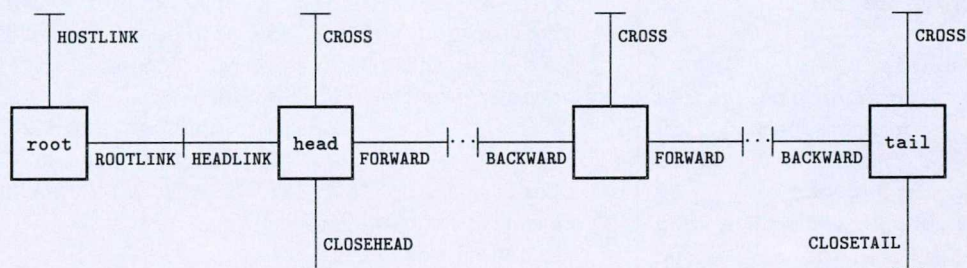


Figure 5.1: Commune processor network

The `LIDX.*` (Link InDeX) constants (□172–□180) are declared in accord with the T.Node requirements and with the ring to be built. Figure 5.1 explains the link index names. `NofNodes` (□182) is set to the number of commune components, this is the only constant that has to be redefined if we want to use different number of processors. The hardware network description (□186–□225) declares the processors, sets their attributes and connects the links appropriately. The software network description (□231–□275) declares the processes and the channels, then connects all these appropriately. Care must be taken to ensure that channels in the Hamilton cycle get in the appropriate slots in the interface chan-

nel arrays (§ 273). The mapping section is simple, each processor gets one process starting at low priority.

After configuring the TNode machine, the system is started by issuing the command

```
irun· objT8/commune.btl· <description-file>
```

from the base directory. <description-file> is a text file containing runtime parameters for the commune.

The commune system supports different levels of reconfigurability. At the lowest level, the load balancing algorithm can be changed by rewriting some of the code, probably involving `boss/feeder.occ`, `guard.occ` and files from the `store` directory. On the next level, an application process (evaluator) can be plugged in to benefit from automatic tasks distribution. These modifications require rebuild and reconfiguration. One level higher, the processor topology can be changed by rewriting `config/commune.prg`. After such a change, no recompilation is necessary, only the configurer and the collector has to be invoked. At the top level, the following data can be specified on the command line:

- ▶ A file containing initialisation data to the evaluators. For example, the encoded task tree is given when the default application mimic is used, or the compiled program to the PROLOG interpreter. The contents of this file is sent to every evaluator in the initialisation phase.
- ▶ A file containing the task to be evaluated. For example, it may contain a query to the PROLOG interpreter. This task is given to the first processor to start the computation phase.
- ▶ A number defining how much memory should be used to collect runtime profile data. The number 0 disables profiling. If the assigned memory is used up, subsequent profile events are discarded.
- ▶ A number defining how much memory should be assigned to the evaluator.
- ▶ A number defining the maximum size of tasks.

Based on the input data, memory blocks will be allocated to various processes (see below), the remaining memory is assigned to the `store` process. The total amount of memory is set in the configuration file (§ 203, § 205). Those who know the static nature of the OC-CAM language may appreciate these features.

5.2 Initialisation

When the system starts, a number of things happen before the evaluators can begin to process tasks. First of all, the `commnroot` process welcomes the user (§ 497) then goes on to check the description file. The `commnpars` procedure in `commnpars.occ` (§ 536) reads and interprets the contents of the file. If there are no errors, the procedure sets the values of several variables passed to it as parameters:

<code>ProfileSize</code>	Amount of memory for profile data
<code>EvalSize</code>	Amount of memory for the evaluator
<code>MaxTaskSize</code>	Size of largest possible task
<code>InitDataSize</code>	Size of the initialisation data file
<code>RootTaskSize</code>	Size of the initial task (file size)
<code>fid.data,fid.task</code>	Files opened for reading
<code>OutDirName,OutDNameLen</code>	Determine the directory where output is to be written (the same as the directory of the descriptor file)

The procedure itself is fairly long but not very interesting, it is included for completeness sake. The `cmmnroot` process continues with checking whether profiling has been enabled or not ([508]) and prints the parameter values to the console. Then it calls `ProfSetup` ([337]) which is a procedure that checks whether the Hamilton cycle exists. It does this by trying to send the value of `ProfileSize` through the cycle. It expects to receive the number of nodes encountered on the cycle. If there is no cycle, the system hangs. If the number of nodes counted does not equal the number given in the configuration file, then execution terminates in failure. When `ProfSetup` returns, `phase2` is called ([401]).

When processing terminates ([442]), two output files are produced per processor. One contains the collected results ([445]) in binary form as they were taken from the evaluators. These files may require some post-processing to extract information. The other file contains the event records which accumulated at `profServer`. The analyser tools of the test system can be used to process data in these files.

A `ProfSetup` procedure can also be found in `profile/profile.occ` ([1109]). It is called ([819]) in the `commune` process right after it starts. It communicates with the neighbours to drive the check data through the Hamilton cycle. It also takes part in determining correction factors for profiling (see below) to work correctly ([1159]). When `ProfSetup` returns, the free memory is split into two parts, `profBuffer` ([832]) is set as prescribed in the description file, `cmmnData` ([836]) will get the rest.

5.3 Profiling

The profiling functionality provided by the `commune` system is the basis on which the analysis of the runtime behaviour of the application can be carried out. The basic idea is that a profiler process (`profServer`, [1244]) is run in parallel with the other processes. This process is a server, it can handle any number of connected clients and records important *events* occurring during processing. Events are signalled by the clients in the form of messages. The messages are tagged values as is defined by the `profProtocol` protocol ([1235]). The tags indicate what type of event occurred, the interpretation of the attached value depends on the type of event. The server itself only stores the data related to the events, so any encoding scheme can be used without the need to change the server code. Four types of events are predefined:

<code>EVENT.START</code>	Marks the start of application processing
<code>EVENT.END</code>	Marks the end of application processing
<code>EVENT.MARK</code>	Used to place data into predefined positions in the event stream
<code>EVENT.MASS</code>	Used to signal that the number of waiting tasks changed

When a message arrives from one of the clients ([1367]), the `RecordEvent` procedure ([1281]) adds a new item to the stream of collected events. The data supplied by the client

are completed with a *timestamp*. The time is read from a TIMER local to the server (1320). The event data are stored in a compressed format to ensure effective use of available memory.

5.3.1 Clock skew

When analysing the data collected by different processors, we have to determine which events occurred at the same (global) time. This is surprisingly difficult. There are two problems related to transputer clocks. First, the clocks are not started at the same instant, this results in different time values acquired by simultaneous readings on different processors. This problem can be handled by forming a circle and calculating the difference of the clocks at each neighbouring processor. The differences allow us to compute the values with which each processor's clock readings have to be adjusted. Unfortunately, this is not enough. The clocks are not only shifted, but they actually run at different speeds. This effect is known as *clock skew* and has hardware origins.

For example, in the T.Node [20], transputers are plugged in on panels, each containing 8 processors. Each panel has its own oscillator generating signals to its transputers. Transputer clocks on the same panel run at the same speed, but they may differ on different panels and this effect is indeed noticeable. Other hardware configurations may have different characteristics. In any case, the test system is designed to be independent of the transputer board used, so a general solution to the clock skew problem must be sought.

To solve the problem we need to compute not only the offsets between pairs of processors, but also the ratio of their speeds. These two numbers are enough to compensate for the undesirable effects. The ratio is acquired by calculating the difference twice, first at the beginning of the computation, then at the end. Assuming that the speed of every transputer clock is constant, the two measurements give the required ratios.

5.3.2 Computing clock differences

The difference between the clocks of two neighbouring processors are computed in the ProfDiff procedure (1074). One of the processors must execute ProfDiffActive (1078), the other ProfDiffPassiv (1087). The inp and out channels must be placed onto the link connecting the two processors, inp connected to out on the other side and vice versa. Both processes declare their own TIMERS, then perform the following sequence of commands in lockstep:

Active		Passive	
out	! 0	inp	? dummy
Clock	? TimeNow	Clock	? TimeMine
out	! TimeNow	inp	? TimeNext

At the end of this sequence, TimeMine and TimeNext will contain the clock values of the two processors as they were at the same instant. The processes must be run on high priority to avoid interrupts (1096).

5.4 Startup

After initialisation is complete, the system uploads the runtime parameters (402), the initial task (426) and the data file destined for the evaluators (432).

Before these can reach their destination, the `commune` process has to split itself into two (842), spawning the high priority `profServer` (see Section 5.3) and the low priority `cmmnmain` (871). `ProfServer` has three incoming client channels (840), these will be assigned to subprocesses of `cmmnmain`. When `cmmnmain` terminates, it sends a termination message to `ProfServer` (853), then `profClose` (1164) measures the clock differences once again (1230) and sends the accumulated data to the host by calling `ProfCollect` (1176).

`Cmmnmain` reads the runtime parameters (971) and inserts them into the event stream (973) to make them available for the analyser tools. Based on the sizes of the various components, memory is further divided to prepare for subsequent process splits. `Cmmnmain` will split itself to give birth to the three processes known as the `commune` components (see Figures 3.4 and 4.5). The balancer is now called `sentinel` and the worker is called `guard` for historical reasons.

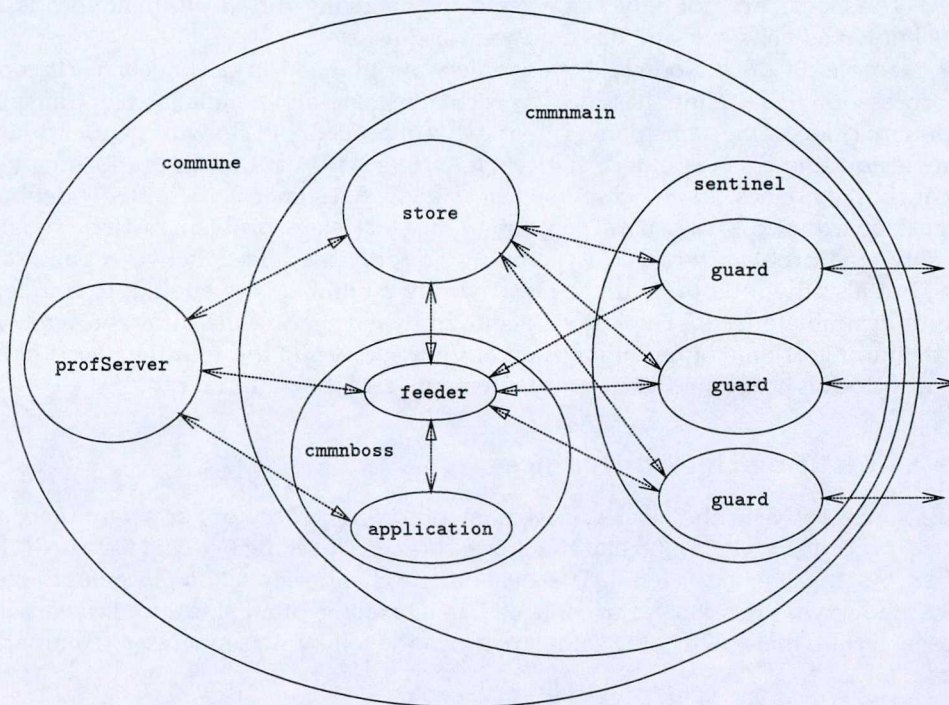


Figure 5.2: Commune process hierarchy

5.4.1 Process hierarchy

As the system progresses towards task evaluation, the first `commune` process is repeatedly split into parallel subprocesses, some of which we encountered already. Figure 5.2 illustrates the final hierarchy of processes. `Store`, `cmmnboss` and `sentinel` are started by `cmmnmain` (1033) after it allocated the memory blocks and internal channels for them (978–1013). `profServer`, `store` and `sentinel` run at high priority, the `cmmnboss` process on low priority. This arrangement makes it possible that load balancing activity can be performed and recorded in preference to application processing.

The processes use the client-server rules to interact with each other, this is emphasized with the triangle arrowheads. We can see that the feeder process is the ultimate client, it schedules work for the application and initiates load balancing actions.

5.4.2 Distribution/collection method

The Hamilton cycle is used to distribute and collect data. The nodes are visited one by one, this can be carried out more easily if suitable channel abbreviations are used.

When distributing data, values traverse the Hamilton cycle backwards. First, the *master* node passes the data to the head, it passes them to the one before it (ie. to the highest numbered node) and so on until the data reaches the node with *id* = 1. It should not send them further because by that time all nodes have been visited. This node is called the *tail* during the distribution phase (879). The node with *id* = 1 was selected because otherwise we would have to acquire the total number of nodes which is not impossible but unnecessary. This decision forces us to traverse the cycle backwards.

Two channels are needed at every node, one to read the data from and another to send them away. With regard to the input channel, head is special because has to read from its environment channel coming from the *master* node (ie. from *cmmnroot*). The others have to read from the 'forward' side. The output channel is the 'backward' channel on the cycle. These channels get determined (890), after which each node first reads (895) then (except for tail) sends the data (899). This way the nodes will be visited sequentially. The scheme of a process participating in distribution is shown in Figure 5.3 (see also *getSizes*, 888).

```
PROC distrib( VAL INT id, [CHAN OF DATA inp, [CHAN OF DATA out, CHAN OF DATA envinp)
.. VAL HEAD IS ( id = 0):
.. VAL TAIL IS ( id = 1):
.. dout IS out[ HamBwSide]:
.. dinpv IS [ inp[ HamFwSide], envinp]:
.. dout IS dinpv[ INT HEAD]: -- INT FALSE = 0, INT TRUE = 1
.. DATA x:
.. SEQ
.. dinpv ? x
.. -- store x locally if needed
.. IF
.. .. TAIL
.. .. SKIP
.. .. TRUE
.. .. dout ! x
.. -- distrib -----
```

Figure 5.3: Distribution scheme

Data collection is similar. The nodes form a conveyor belt on which results are transported to the head and eventually to the host with the help of *cmmnroot*. Again, two channels are needed at every node, one for input (937) and one for output (935). *Head* is special in this case, it needs an extra output channel (936) to initiate the collection process. Each node reads in an integer (946) first which is the size of data packet it is going to receive from its neighbour. The number -1 signals the end of the stream. The node sends its own data (954), then reads in the next packet if necessary (962). The whole process is initiated by the *head* pushing a -1 into the chain (943). The collection scheme is captured by the process declaration in Figure 5.4 (see also *sendResults*, 932).

token at termination detection. Next, available memory is divided into *slots* (1465). One slot holds one task plus two more fields, a *link field* containing the slot index of the next element in the list and a *size field* holding the true size of the task. The variables *nofUsed* (1468) and *nofFree* contain the number of slots belonging to the used and the free list respectively. The variable *firstUsed* and *firstFree* contain the slot index of the first elements of the lists.

A client of the store can be in a *waiting* state, meaning that it wants to be informed if the state of the store changes. The *waitFlag* array (1477) is used to remember which client is waiting. Two more arrays, *slots* (1483) and *bookFlag* (1485) store information about the bookings of each client. A number of procedures and functions are defined (1494–1534) to handle the lists.

When store starts (1793), it executes an initialisation procedure *init* (1536) to set the various state variables. Then it enters its server loop, listening to incoming channels (1757) and servicing the requests as appropriate. After each request is serviced, it notifies the waiting clients.

5.6 Application

The application process is defined as a server of the feeder process (Figure 5.2). The code listing in the appendix contains a “dummy” application (1948) that generates tasks by using the random number generator. The application has to accept a task, process it, and report the outcome. The protocols *APP.REQUEST* and *APP.ANSWER* in *app/appifac.inc* (1928) define the message formats precisely. The evaluator initialisation data the user specified in the description file on the command line is given to the application by using the *ar.init* tag (1930). This happens before normal processing begins giving an opportunity to the application to initialise its internal data structures if necessary (2011). During processing, the *ar.work* tag is used to give a task to the evaluator (2016). If the evaluator chooses to decompose the task, it sends the descendant tasks in *aa.decomp* messages (1989) to the feeder. When it finished with the *input task*, it replies with an *aa.ready* message. The evaluators have to accumulate all the output data they produce during processing. The tag *ar.close* message tells the evaluator that there are no more tasks, it has to reply back the accumulated results using the *aa.results* tag. Finally, *ar.quit* tells the evaluator to terminate.

The *ar.status* and *aa.status* pair of tags can be used to tighten the interaction between the feeder and the application. While working on a task, the application can send reply messages tagged with *aa.status*. The feeder “replies” with a boolean value indicating whether task decomposition is preferable. It says *TRUE* if the store is low, *FALSE* otherwise. The use of this feature helped to achieve good performance with the pure PROLOG interpreter.

When started, the application is supplied with a channel leading to the *profServer* process, it can use this channel to record events as it sees fit.

5.7 Feeder

The feeder and application processes are launched (2066) by the boss after it prepared the channels and memory blocks for them. The feeder can be found in the file *boss/feeder.occ* (2082) and encapsulates most of the processor commune functionality.



It starts (2308) by creating a termination token at the head node (2312). Initialisation data is sent to the evaluator (2316) and the main loop is entered. It consists of two phases, a task processing phase and a termination detection phase. The two phases alternate until termination is detected. The `processtasks` procedure (2113) schedules tasks for the evaluator and receives results until the store becomes empty. When there are no more tasks, it returns. The `feeder` then broadcasts a request to all of its neighbours, activating the load balancing components (2329). The `termindetect` procedure (2241) is called next to determine whether computation has finished on all nodes. It checks whether the termination token is in its local store. If it is there, the new token color is computed and the token is sent away (2275). The token circulates on the Hamilton cycle until the head finds that it completed two cycles without encountering any activity. Then the rest of the nodes are notified and the `feeder` terminates. This triggers the results to be retrieved from the evaluators in the boss (2075), the results to be collected (1072) in `commmain` and eventually the termination of all the remaining processes.

5.8 Load balancing components

The `sentinel` process (2335) is started by `commmain` to take care of the interaction with neighbouring processing elements. Its only task is to spawn guard processes, one for each transputer link. Each guard is responsible for the communication on its assigned link. They will occasionally have to transfer task records from one node to the other, so `sentinel` allocates buffer space for each of them.

The guard process is essentially the same as the GDI worker from the previous chapter (Section 4.6). `SentinelProtocol` (2393) corresponds to the TRANSACTION protocol, the transfer procedure was left undefined, now it can be found (2419) in its full generality. We now have subtypes of control data, this induces changes to the program structure compared to the one given in the previous chapter.

Chapter 6

Node processing model

Based on [9].

6.1 Introduction

Parallel programming languages have two advantages over sequential ones. One is that a parallel program can run faster if suitable hardware is available, the other is that as programs often model real world phenomena, the solution may be expressed more naturally using explicit or implicit parallelism. Despite this potential, parallel programming has not become widespread because there are no general purpose parallel programming models (see also [27], [28]). It is important that we make a distinction between architectures and programming models. A programming model is an abstract machine with its data structures and operations that connects the physical machine and the higher level language(s) used by humans. It has to meet conflicting requirements from the two sides, it has to be convenient to use for problem solving and has to allow efficient implementation(s). Humans are given in this scenario, the task is to find a model and design a machine such that the three together satisfy the conditions above. A number of parallel programming models and parallel architectures have been proposed, some are based on the successful sequential von Neumann model and architecture while others are based on new ideas. The von Neumann model has global random access memory as its data structure, the operations fetch, process and store data sequentially.

A parallel architecture consists of processors (or processing elements, PEs), memory module(s) and optionally some kind of interconnection mechanism to connect PEs and/or PEs and memory modules. The most popular classification of architectures distinguishes four main classes: SISD, MISD, SIMD and MIMD, the last two having shared memory (SM) and distributed memory (DM) versions. A general purpose parallel architecture has to be extendible, that is users have to be able to add PEs and memory to increase processing power. An ideal general purpose parallel programming model on the other hand has to hide the architectural details from programmers. They must be protected from having to deal with such details as the number of PEs, interconnection topologies, or placement of processes to processing elements. If these requirements are satisfied, portable parallel programs can be written, opening the way to widespread use of parallel computing. Note that portability does not mean that programs could be run on every machine, it only means that they can be run on the single architecture (or few architectures) which accompany the general programming model. As such machines are extendible, their speed and capacity

can be adjusted by the users to their needs.

The NOP model proposed here incorporates a number of novel features. It has a single data structure that makes random memory accesses unnecessary. This radical approach has both advantages and disadvantages, but it is argued that it is worth considering as a viable alternative method of memory management in the context of parallel processing. The model has both imperative and declarative features, trying to combine the best of both worlds. Its simple imperative transformations can be easily and efficiently implemented, while larger programs can be constructed in a declarative style, benefiting from the substantial body of accumulated knowledge in the area of functional programming languages. By studying the existing programming models (see Chapter 1), we can observe that random access causes implementation difficulties in distributed and even in shared memory environments. A more regulated, more predictable flow of data would be more satisfactory. Further, the imperative style of programming is preferable if we want performance, declarative style is better at ensuring sound design and program correctness.

The NOP model uses implicit references exclusively, it is not possible to use variables. Operations cannot name their arguments, thus making random access impossible. The operations are commands in the imperative sense, they cause some action(s) to be performed. There are sequential, conditional and parallel composition operations. On the other hand, a NOP program is a mutually recursive equation set, in this sense it is declarative. Put into other terms, a functional program (equation set) determines the control structure of the imperative program, which is then executed to map the input into the final result. Using equations, programmers extend the transformation set of the model. These design choices are believed to produce a unique, radical, but satisfying model.

The ultimate goal of the model is of course to allow efficient multiprocessor implementations. A fully functional NOP simulator has been developed and implemented to carry out experiments and gain insight for further research. This sequential NOP simulator immediately translates into an SM-MIMD implementation, the only problem to solve is to regulate access to global data, but this can easily be done using standard mutual exclusion techniques. Research progresses toward investigating a combination of program transformations and dynamic load balancing to arrive at a distributed memory multiprocessor implementation.

6.2 The NOP model

6.2.1 Data

The only data structure in the NOP model is the *node*. A node can be thought of as a memory cell which is capable of holding a *value*. Apart from the value, a node can have an arbitrary number of *subnodes*, each of which is a node like itself. The subnodes form a bidirectional ring, so it is possible to speak of the left or right neighbour of any given subnode. When there is only one subnode, it is the left and right neighbour of itself at the same time. If a node has at least one subnode, then one of them is the *active* or *selected* subnode. The active subnode has a special role in certain transformations. In general, a node is the root of a tree of nodes with its subnodes being its immediate descendants in the tree. No matter what the size of the tree is, only the root node and its active subnode are directly accessible. Values that are stored in node cells can be *atoms* or nodes. Atoms are indivisible entities, they have no builtin properties other than they are different from each other. There is an atom called *null*. By allowing nodes to contain other nodes as values, a certain 'multi-tree' structure can be built. Atoms are denoted by strings of characters (*null*,

x , xxx , $b5$), a node with value V and subnodes N , O and P is denoted by $V(N O P)$, for example $x(a() b())(y(w()) z())$ is a node with value $x(a() b())$ and subnodes $y(w())$ and $z()$. The active subnode is always the first in the subnode list.

In graphical representation, node cells are depicted as rectangles. The value of the node may appear pointed to by an arrow leading out of the rectangle, or atoms may be placed inside the rectangle. The subnodes are placed in order below the node and connected to it, the active subnode being indicated by a filled arrowhead.

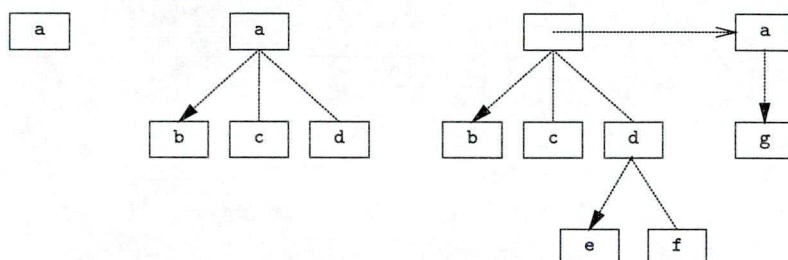


Figure 6.1: Graphs of nodes

Examples of node graphs (Figure 6.1): $a()$ has value a (an atom) and does not have any subnodes. $a(b() c() d())$ has value a and has subnodes $b()$, $c()$ and $d()$. The node $a(g())(b() c() d(e() f()))$ has value $a(g())$ (a node) and has subnodes $b()$, $c()$ and $d(e() f())$.

Empty pairs of parentheses can be omitted without causing ambiguity if the value of a subnode is an atom.

$$\begin{aligned} a(b() c() d()) &\leftrightarrow a(b c d) \\ a(g())(b() c() d(e() f())) &\leftrightarrow a(g)(b c d(e f)) \end{aligned}$$

Finally, a list of single character atoms can be grouped together by using quotation marks.

$$\begin{aligned} a(b c d) &\leftrightarrow a("bcd") \\ a(g)(b c d(e f)) &\leftrightarrow a(g)("bc" d("ef")) \end{aligned}$$

Operations in the model (called *transformations*) work on an implicit node passed to them. Direct access is allowed only to the root of the node tree and to its active subnode. Other subnodes are accessible only after suitable preparations, effectively a "walk" (transformation sequence) is needed to bring the desired node into position. This may seem to be inconvenient and inefficient, but experience shows that these "walks" can be made convenient by hiding them into suitably defined transformations. Also, experiments with a simulator written for the model suggest that the inefficiencies are tolerable.

6.2.2 Parallelism

The most important feature of the model is how it handles parallelism. There is a transformation for this purpose, called *a11*. It has a parameter, a transformation which is applied to all subnodes of the given node. More precisely, let us denote by $\{X, N\}$ the result of transformation X applied to node N . Then

Description	Mnemonic	Definition	Cost
No operation	skip	$\{\text{skip}, N\} = N$	1
Set value	set(V)	$\{\text{set}(V), W(\dots)\} = V(\dots)$	$1 + \text{size}(V)$
Exchange values	exch	$\{\text{exch}, V(W(\dots)\dots)\} = W(V(\dots)\dots)$	1
Copy up value	cup	$\{\text{cup}, V(W(\dots)\dots)\} = W(W(\dots)\dots)$	$1 + \text{size}(W)$
Copy down value	cdn	$\{\text{cdn}, V(W(\dots)\dots)\} = V(V(\dots)\dots)$	$1 + \text{size}(V)$
Select left subnode	left	$\{\text{left}, V(N \dots O P)\} = V(P N \dots O)$	1
Select right subnode	right	$\{\text{right}, V(N O \dots P)\} = V(O \dots P N)$	1
Insert subnode	insert	$\{\text{insert}, V(N \dots O)\} = V(\text{null}() N \dots O)$	1
Delete subnode	delete	$\{\text{delete}, V(N O \dots P)\} = V(O \dots P)$	1
Value from node	split	$\{\text{split}, N\} = N()$	1
Node from value	replace	$\{\text{replace}, N(\dots)\} = N$ N is a node	1
		$\{\text{replace}, A(\dots)\} = A()$ A is an atom	1

Table 6.1: Primitive transformations

$$\{\text{all}(X), V(N_1 N_2 \dots N_k)\} = V(\{X, N_1\} \{X, N_2\} \dots \{X, N_k\})$$

The advantage of this parallel operation is that it is initiated on disjoint data sets, so the resulting “processes” can work independently. The parallel operation terminates when all subnodes have been transformed. To initiate parallel processing, the program first has to create independent groups of data (subnodes of a node). Again, the need to rearrange data admittedly costs programmer effort and computing resources, but experience with the simulator suggests that by defining suitable high-order transformations, programs can be clear and easy to write. At the same time, the use of computing resources during rearrangement is offset by the ease of scheduling the independent “processes” that result from the initiation of the all transformation.

Description	Mnemonic	Definition	Cost
sequence	seq(X Y)	$\{\text{seq}(X Y), N\} = \{Y, \{X, N\}\}$	$1 + C(X, N) + C(Y, X, N)$
conditional	equ(X Y)	$\{\text{equ}(X Y), N\} = \{X, N\}$ if $N = V(V(\dots)\dots)$	$1 + \text{size}(V) + C(X, N)$
		$\{\text{equ}(X Y), N\} = \{Y, N\}$ otherwise	$< 1 + \text{size}(V) + C(Y, N)$
parallel	all(X)	$\{\text{all}(X), V(N_1 N_2 \dots N_k)\} = V(\{X, N_1\} \{X, N_2\} \dots \{X, N_k\})$	$k + C(X, N_1) + C(X, N_2) + \dots + C(X, N_k)$

Table 6.2: Compound transformations

6.2.3 Transformations

There are primitive transformations that work with nodes in elementary ways (Table 6.1). Remember that the particular node a transformation transforms is implicit. There are three builtin first-order (compound) transformations (Table 6.2), these take zero-order transformations as parameters. The ellipsis in each definition is intended to stand for zero or more nodes, obviously for the same ones on both sides in the same position. The equ transformation uses its first parameter to transform the (implicit) node if the value of that node

equals the value of the node's active subnode. Otherwise, it uses the second one. Seq is self-explanatory, all has been discussed earlier. We will come back to the cost column in the implementation section.

6.2.4 Discussion

A number of questions have to be answered about the model:

- ▶ Is it capable of computing every computable functions?
- ▶ Which architecture(s) are suitable for it?
- ▶ Can it be used as a basis to define high level programming languages?
- ▶ Are these languages adequate for large-scale programming projects?

Not all of these questions can be answered completely because more work is needed to resolve them. A fully functional NOP simulator has been developed and implemented to carry out experiments and gain insight for further research.

Completeness

Completeness of the model can easily be shown by constructing a NOP transformation that simulates a universal one-tape Turing Machine (TM). The construction is relatively straightforward. The state and tape of the TM is represented by a one-level node structure (root and subnodes), the actions of the TM are easily reproduced by simple NOP transformations. It is worth mentioning that the following NOP transformations are not required for the simulation: `cdn`, `cup`, `split`, `replace`, `all`. All is not surprising, as the TM is a simple sequential device, it really should be possible to simulate it without using parallelism. The first four are included in the model so that we could write more efficient programs than it is possible with TMs. The `split` and `replace` transformations were included in the model to enable us to rearrange the node structure efficiently. This is possible because their cost is constant, but by using them we can "cut and paste" whole subtrees of nodes.

Efficiency

In terms of speed, the NOP model may seem to be inherently as weak as a Turing machine because the focus of computation can only be shifted gradually from one place to another. In fact the NOP model offers the same efficiency as a logarithmic-cost RAM [33]. RAM operations can be considered as transformations that work on the whole of the global memory, the part affected is identified by an address. This behaviour can be simulated in the NOP model by representing global memory with a suitable node (a binary tree) and RAM operations with equivalent transformation sequences. The depth of the binary tree is equal to the number of address bits used by the RAM. When simulating a random access, the NOP transformations have to "walk" to the particular leaf identified by its address. This way, the time to simulate each random access is proportional to the number of address bits. Assuming that transformations equivalent to RAM operations exist, any RAM algorithm that uses W address bits to access its data can be simulated by a NOP program that runs W times slower.

6.3 Programs

A NOP program is a potentially infinite (compound) transformation. Infinite transformations occur when iteration is needed to express an algorithm. Of course this is only an abstraction; programs can conveniently be specified finitely using recursive equations. Let us see an example where an iterative algorithm is needed. Suppose that we are given a node of the following form:

$$A(A_1() A_2() \dots A_k()) \quad \text{where } A \text{ and } A_i \text{ are atoms}$$

We are required to construct a transformation that makes active a subnode whose value is the same as that of the node, assuming that there is such a subnode. First we should check whether $A = A_1$. If so, nothing else should be done. If $A \neq A_1$, then we select the right neighbour of the active subnode and carry out the same check, and so on until we find a match. Let us call the transformation that works like above `locate`. Then

$$\text{locate} = \text{equ}(\text{skip seq}(\text{right locate}))$$

specifies how `locate` should work. This is an infinite compound transformation represented finitely. Based on this observation, a program in the NOP model is a set of mutually recursive transformation definitions, where each definition introduces a unique transformation name.

The primitive transformations in the NOP model are very low level operations, they comprise the “machine language” of the NOP abstract machine. Any programming language that is based on the NOP model defines node transformations in some way. The particular language introduced here does this by extending the notation used so far.

We will use strings of letters, numbers and underscores as transformation names and separate the two sides of definitions by colons instead of equal signs. The form of compound transformations as defined above are often inconvenient to use, therefore we introduce operator forms below in order of decreasing precedence.

$$\begin{aligned} @X &\leftrightarrow \text{all}(X) \\ X;Y &\leftrightarrow \text{seq}(X Y) \\ =X\&Y &\leftrightarrow \text{equ}(X Y) \\ X?Y &\leftrightarrow \text{equ}(X Y) \end{aligned}$$

$$\text{Thus, for example } @X ? @Y; Z \leftrightarrow \text{equ}(\text{all}(X) \text{seq}(\text{all}(Y) Z))$$

Sequential composition is associative, so

$$\begin{aligned} X;Y;Z &\leftrightarrow \text{seq}(X \text{seq}(Y Z)) \\ X;Y;Z &\leftrightarrow \text{seq}(\text{seq}(X Y) Z) \end{aligned}$$

A sequence of two or more transformations will also be denoted by $[X Y \dots Z]$ which is equivalent to $X;Y;\dots Z$.

Direct nesting of `equ`s is not particularly useful, because

$$\begin{aligned} \text{equ}(X \text{equ}(Y Z)) &\equiv \text{equ}(X Z) \\ \text{equ}(\text{equ}(X Y) Z) &\equiv \text{equ}(X Z) \end{aligned}$$

so $X?Y?Z \equiv X?Z$ where $X \equiv Y$ means $\{X, N\} = \{Y, N\}$ for all nodes N . Now the `locate` transformation can be defined as

$$\text{locate: skip ? [right locate]}$$

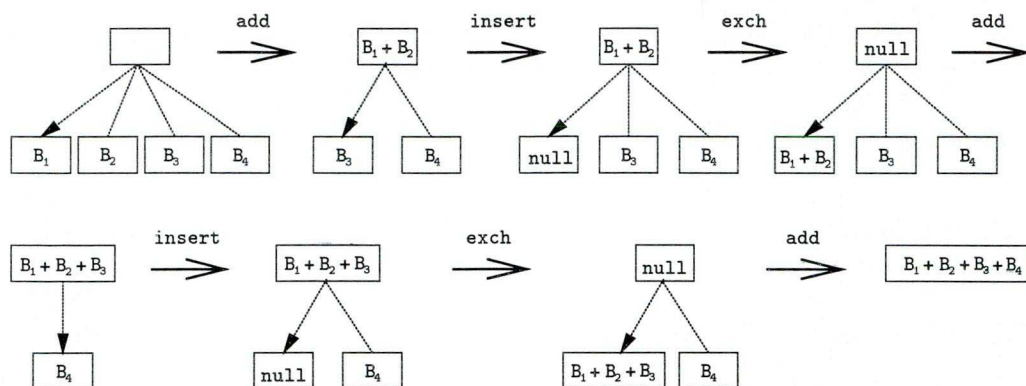


Figure 6.2: Sequential add4

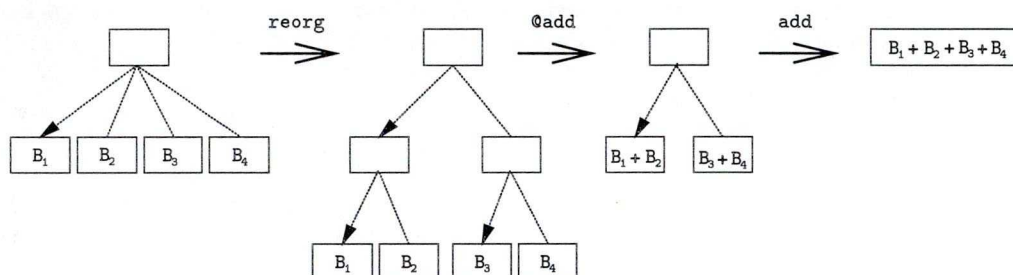


Figure 6.3: Parallel add4

The notation introduced here serves as a basis for a higher level language based on the NOP model.

6.3.1 Example

Let us see a simple example. Assume we represent binary digits with the atoms 0 and 1, binary numbers with nodes that contain binary digits in their subnodes. The value of the node itself is not significant. For example, the number 5 can be represented by `null("101")`. Further, suppose we have a transformation called `add` that adds two numbers, more precisely

$$\{\text{add}, v(B_1()) B_2() N*)\} = B(N*)$$

where B_1 , B_2 , B are binary numbers and B represents the sum of B_1 and B_2 .

We would like to have a transformation `add4` that adds 4 numbers, ie.

$$\{\text{add4}, v(B_1() B_2() B_3() B_4())\} = B()$$

where B is the desired sum. A straightforward sequential solution might be (Figure 6.2):

```
add4: [ add addnext addnext]
addnext: [ insert exch add]
```


We might do better, however, if we add two pairs of numbers in parallel and then add the results to form the final answer. For this to be possible, we first have to reorganise the four numbers into two pairs.

```
add4: [ reorg @add add]
```

Reorganising transformations lie at the heart of NOP programs. Figure 6.3 shows in graphical form how the transformation works. The reader is advised to construct such diagrams for later examples.

6.3.2 High level NOP language

A NOP-based high level programming language provides facilities for programmers to define transformations. The most natural way to do this is by extending the notation defined for NOP programs earlier. In fact such notations exist, they are called functional languages. Looking at them from a functional perspective, primitive NOP transformations have type $N \rightarrow N$, where N is the set of nodes. (set is an exception, its type is $V \rightarrow N \rightarrow N$). Compound transformations have the following types assuming $T = N \rightarrow N$:

```
seq: T → T → T
equ: T → T → T
all: T → T
```

Based on these functions, the full power of functional languages can be incorporated into defining transformations. The execution of such a high level language program is accomplished in two phases. First the functional program is executed to result in function of type T , then this function containing only the transformations defined in the NOP model is executed by the NOP executor. If we apply a lazy functional language, then the two phases can even overlap, the functional phase supplying food for the NOP phase as demanded. Before execution of a NOP program, we can carry out optimisations on it, making the program easier to parallelise and therefore run faster. This is achieved by recognising that the equation

$$\text{seq}(\text{all}(X) \text{all}(Y)) \equiv \text{all}(\text{seq}(XY))$$

holds for any transformations X and Y . This means that all transformations can be migrated upwards in the transformation tree representing the program, this way making its granularity coarser. Fortunately, coarser granularity programs are easier to parallelise using dynamic load balancing techniques (see Chapter 3) further strengthening the viability of the NOP model.

6.3.3 Parameterised transformations

The most important feature that we need from functional languages is the ability to generalise declarations by introducing parameters. Consider the `locate` example. If we are looking for the atom `a`, we can write

```
locate_a: [ 'a' locate]
```

Using this method we can write as many `locate_*` transformations as we want, but a more flexible way would be better. We can achieve this by using parameterised transformations of the form

```
locate( WHAT): [ WHAT locate]
```


Here WHAT stands for an arbitrary transformation that sets the value of the node to what we are looking for. References to it must specify an argument to be used in place of WHAT, so if we insist on giving a name to it, `locate_a` can be defined as

```
locate_a: locate( 'a')
```

The method used here is generalisation and specialisation. From many concrete instances of transformation definitions a more general one is created with the introduction of formal parameters. Specialisation is the opposite, a general form is used together with arguments to yield a concrete instance. The use of a transformation name with different number of arguments is allowed, as long as every name/arity combination occurs only once. Note that compound NOP transformations can be regarded as parameterised transformations (`seq/2`, `equ/2`, `all/1`). Other parameterised transformations:

```
const( NODE): [ NODE replace]
```

`const/1` is the "constant" transformation. The output node is always `NODE` regardless of what its input is. For example the output of

```
[ const( 'null( "abcdefg")') locate( 'c')]
```

is `c( "cdefgab")`.

```
seta( VALUE): [ exch VALUE exch]
```

`seta/1` assigns `VALUE` to the active subnode leaving the value of the node unchanged. `VALUE` is expected to stand for a 'set value' transformation (eg. `'a'`) and there must be at least one subnode. Note that these additional restrictions cannot be expressed in the notation.

```
setr( VALUE): [ right seta( VALUE) left]
```

`setr/1` is the same as `seta/1` but it **changes** the value of the right neighbour of the active subnode.

```
dor( X): [ right X left]
```

`dor/1` does not need explanation. Similar definitions are

```
dorr( X): dor( dor( X))
```

```
dol( X): [ left X right]
```

etc. Now `setr/1` can be redefined as

```
setr(VALUE): dor( seta( VALUE))
```

Similarly

```
setl( VALUE): dol( seta( VALUE))
```

A frequently needed transformation is the following

```
insert( V): [insert seta( V)]
```

it creates a new (active) subnode with value `V`.

A generalised version of comparison is “constant” equ

```
cequ( V Y N): [ V =Y & N]
```

We will often need a construct similar to the CASE statement of pascal and similar languages.

```
case( V MATCH NOMATCH):  
  [insert( V) =[ delete MATCH] & [ delete NOMATCH]]
```

case/3 is similar to equ/2 in that it uses one of two transformations to transform its input node. MATCH is invoked if the value of the node is the same as V, otherwise NOMATCH is used. Note that when MATCH or NOMATCH is invoked, the node they get has to be the same as that given to case/3, this is why delete occurs in front of both MATCH and NOMATCH. More choices can be offered by

```
case( V1 M1 V2 M2 N): case( V1 M1 case( V2 M2 N))
```

The catch-all parameter is not always needed:

```
case( V1 M1 V2 M2): case( V1 M1 V2 M2 skip)
```

Similar to case/3 when we want to base our decision on the value of the active subnode

```
casea( V M N): [ exch insert( V) =[ delete exch M] & [ delete exch N]]
```

case/3 and casea/3 show the same pattern, namely the node has to be prepared for a test, then after the decision has been made, the node has to be brought back to its initial state before transforming further. This pattern is expressed by

```
doequ( BEFORE AFTER SUCCEED FAIL):  
  [ BEFORE =[ AFTER SUCCEED] & [ AFTER FAIL]]
```

Again it is the responsibility of the programmer that [BEFORE AFTER] is equivalent to skip. Now case/3 and casea/3 can be redefined as

```
case( V M N): doequ( insert( V) delete M N)  
casea( V M N): doequ( [ exch insert( V)] [ delete exch] M N)
```

6.3.4 Transformation name parameters

Generalisation on complete transformations is not the only way we can extend our language. Parameters can be introduced to be used in place of transformation names, this way the “called” transformation is told what other transformation to use but it is free to pass parameters on its own. Let us consider again the locate example. It is conceivable that checking whether the active subnode has the right value is more complicated than simple equality testing. By introducing parameters that stand for transformation names, it is possible to write this more general locate as

```
locate( Test): Test( skip [ right locate( Test)])
```

It is very common for transformations of this sort to refer to themselves, but if there are a lot of parameters, writing out all of them is tedious and error prone. To avoid such problems

a *star* will be used when a transformation has to refer to itself on the right hand side of the definition.

```
locate( Test): Test( skip [ right *])
```

The original `locate` can now be defined by

```
locate: locate( equ)
```

Merging both parameterised versions yields

```
locate( WHAT Test): [ WHAT locate( Test)]
```

Other useful transformations with name parameters are the NOP equivalents of while and repeat-until loops

```
while( Test BODY): Test( [BODY *] skip)
repeat( BODY Test): [ BODY Test( skip *)]
```

One more redefinition of `locate` is

```
locate: while( neq right)
neq( Y N): equ( N Y)
```

In a number of cases so far we defined parameterised transformations that worked by selecting one of a subset of their arguments to transform the input node. These were `equ/2`, `neq/2`, `case/3`, `casea/3` and `doequ/4`, the latter three of them selects one from their last two parameters. Let us call transformations having this property *selector transformations*, they are important because they are easy to use in compositions. Selectors with two parameters can be thought of as binary decision procedures and as usual we may want to combine these using boolean operations. Fortunately, they are easy to construct with name parameters.

```
not( Test Y N): Test( N Y)
and( TestA TestB Y N): TestA( TestB( Y N) N)
or( TestA TestB Y N): TestA( Y TestB( Y N))
```

Here `Test`, `TestA` and `TestB` are supposed to be names of binary selectors, while `not/3`, `and/4` and `or/4` are themselves selectors in their last two parameters.

Before proceeding further we need to define a few simple but useful transformations that will be used subsequently.

```
insexch: [ insert exch]
exchdel: [ exch delete]
expand: [ split insexch @replace]
contract: [ @split exch replace]
```

Both pairs are complementary in the sense that `[insexch exchdel]  $\equiv$  skip` and `[expand contract]  $\equiv$  skip`. `expand` expands a node by placing a new node at the top with the original node as its only subnode. `contract` does the opposite if used on nodes with exactly one subnode.

Recall that `all/1` works by transforming all subnodes of a node with the same transformation. It is often desirable to use different transformations on different subnodes, so we

define `all/2` that transforms the active subnode with its first, the rest of the subnodes with its second parameter. The first attempt is

```
xall( ACTIVE REST):
    [ @expand seta( 'A') @case( 'A' @ACTIVE @REST) @contract]
```

It first expands all subnodes, this way making the node structure one level deeper, all the newly inserted node cells contain the null atom. After that, it sets the value of the active subnode (which is now one of the new ones created by `expand`) to `A`. Then all subnodes are transformed by `case( 'A' @ACTIVE @REST)` which transforms the original subnodes as needed either selecting `@ACTIVE` or `@REST`. Finally, all subnodes are contracted to remove the extra nodes from the structure. As `seta/1` works correctly only if there is at least one subnode, this also applies to `xall/2`. Another option is that we define the transformation

```
keepval( X): [ expand exch @X exch contract]
```

which saves the value of its input node before a transformation and restores it after. Applying this, `xall/2` can be made safe to use on nodes without subnodes.

```
all( ACTIVE REST): keepval( xall( ACTIVE REST))
```

Occasionally we need to transform only the active subnode:

```
sub( X): all( X skip)
```

These two transformations can be defined in the same fashion with more parameters as well.

6.4 High level programming

A few examples have already been given that illustrate how one can solve computational problems in the NOP language. Now we define new parameterised transformations that are useful for solving more complex problems that involve parallelism. It was noted earlier that reorganising transformations are very important because in the NOP model these are the only means to prepare data to be processed in parallel. Frequently, reorganisation takes place by enumerating the elements of the subnode list and building a different structure. Next we will define transformations that can be used to achieve this.

6.4.1 Enumeration

Enumeration is straightforward if there is a known number of subnodes, but seems more problematic otherwise. To illustrate this, let us consider once again our `locate` example. Now we want it to terminate even when the value looked for is not found. The first idea might be to insert a new subnode with a known value which could be used to check whether all other ones have been considered. The problem with this approach is that we cannot be sure that the extra inserted value does not occur among the others, so the idea does not work in general. A better way to solve the problem is similar to that of used in `all/2`, ie. we expand all subnodes before inserting the marker, then after each shift, check whether the marked subnode has been reached. The nodes created by `expand` can be set to any value we want, so we are able to ensure that the marker is different from its neighbours. We must remember however that the original subnodes are found one level deeper after `expand`.


```

locate: [
  @expand
  prepare
  repeat( right going)
  finish
  @contract
]

prepare: [ @'LOCATE' insert]

finish: casea( null delete [ right * left])

going( Y N):
  casea(
    null N
    )    doequ( sub( exch) sub( exch) N Y)
  )

```

General enumeration works in a similar fashion, it will consist of two transformations, one for preparation and another for enumeration which also restores the original status if all subnodes have been enumerated. To prepare, we define

```
enumbegin: [ @[ expand 'ENUM'] insert]]
```

The main goal of enumeration is that each subnode value is brought up to the root node. This can be done in one of two ways, either by an exchange transformation which removes the original or by a copy transformation that leaves it intact. Let these options be left to the user:

```

enumnext( MODE): [
  right
  nullequ(
    [ delete @contract]
    [ sub( MODE) exch]
  )
]

nullequ( Y N): cequ( null Y N)

```

where `MODE` can be either `exch` or `cup`. Better still, we give a name to both available forms:

```

enummove: enumnext( exch)
enumcopy: enumnext( cup)

```

6.4.2 Producer-consumer pairs

In operating systems and other parallel processing environments, producer-consumer process pairs are used frequently to implement cooperation between two parties. The producer produces items and passes them through a limited size buffer to be consumed by the consumer. When there is no buffer, the two sides synchronise upon data exchange, otherwise proceed in parallel. We will define a transformation that mimicks the behaviour of a bufferless producer-consumer pair. This is achieved by using a node with two subnodes, one of them belongs to the producer, while the other one to the consumer. A producer transformation is used to extract the next value which must be put into the left subnode, then it is moved into the right subnode, which is then transformed by the consumer transformation. The producer signals the end of item stream with the atom *null*. With this separation of functionalities, a high degree of flexibility is achieved.

```

pc( PROD CONS): [
  sub( PROD)
  while( pcgoing [ pcxfer sub( PROD CONS)])
]

```



```
pcgoing( Y N): nullequ( N Y)
pcxfer: [ exch dor( exch)]
```

To see the introduced concepts in action, let us consider the add4 transformation mentioned earlier, where reorg was left undefined.

```
add4: [ reorg @add add]
```

We can use pc to define reorg. The producer will be enummove, providing each number in sequence. The consumer (grouper) will be a transformation that arranges incoming numbers into two groups. As there are four numbers to start with, there will be two two-element groups at the end. The two groups are two nodes with the appropriate number of subnodes with values corresponding to group members.

```
grouper:
  casea(
    'left'  [ exch @[ putitem 'right']]
    'right' [ exch @[ dor( putitem) 'left']]
  )
putitem: [ exch sub( insexch)]
```

Now reorg can be defined as

```
reorg: [
  pcprepare(
    enumbegin
    const( 'item( left( left right))')
  )
  pc( enummove grouper)
]
pcprepare( PB CB): [ expand insert right sub( PB CB)]
```

6.4.3 Divide and conquer

There are a number of problems that can be solved by applying a method called divide-and-conquer, where a problem to be solved is decomposed into subproblems, then after solving them separately the subsolutions are combined into a solution of the original problem. This method is very popular in algorithms written in declarative languages and it is very easy to define a parameterised NOP transformation that serves as a basis for divide-and-conquer algorithms. It has four parameters, Test to decide whether the problem at hand can be solved directly or has to be decomposed, ATOMIC to solve a subproblem without decomposition, DIVIDE to decompose problems and CONQUER to combine subsolutions. The format of the problem nodes it can work on is not restricted, the user must provide transformations in place of the parameters that work consistently with each other. DIVIDE accepts a suitable node and transforms it into one whose subnodes represent subproblems. Conversely, CONQUER expects subsolutions to be present as subnodes of its input node and combines them into a node representing a single subsolution.

```
divide_conquer( Test ATOMIC DIVIDE CONQUER): Test( ATOMIC [ DIVIDE @* CONQUER])
```

6.4.4 NOP quick sort

The main goal of the definition of the NOP model is to enable us to write machine-independent parallel programs. We now demonstrate the use of the model on a “real” problem, quick sort. Sorting algorithms work by comparing pairs of items and rearranging

them based on the outcomes of comparisons. Ideally the comparison function is not built in the sorting algorithm but passed as a parameter, giving the user more flexibility. Our quick sort transformation will have a parameter standing for the name of the comparator which is a selector transformation in all three parameters and is expected to compare the value of the input node and that of the active subnode. The outcome of the comparison can be LESS, EQUAL or GREATER and the comparator has to activate the proper parameter corresponding to that outcome. For illustration purposes we define a comparison transformation that is able to compare atoms A, B and C. By definition A is the least and C is the greatest of them.

```
cmp( LESS EQUAL GREATER):
= EQUAL
& case(
  'A' LESS
  'C' GREATER
  casea( 'A' GREATER LESS)
)
```

Using the building blocks defined earlier, qsort can itself be defined as

```
qsort( CMP): divide_conquer( ifleaf skip qsplit( CMP) qmerge)

qsplit( CMP): [
  pcprepare( enumbegin const( 'new( pivot( gt le)'))
  sub( enummove)
  pcxfer
  dor( @exch)
  pc( enummove qsplace( CMP))
  right
  contract
  contract
]

qsplace( CMP): [
  exch
  CMP(
    @dor( qsp)
    @qsp
    @qsp
  )
  exch
]

qsp: [ exch sub( insexch)]

qmerge: [
  dor( [ exch sub( insexch)])
  sub( enumbegin)
  pc( enummove insexch)
  right
  contract
]
```

As can be seen, qsort/1 uses the divide-and-conquer approach. The items to be sorted have to be present as the values stored in the subnodes of the input node, eg. `unsorted( "ABBACAB")`. The atomicity test is the binary selector `ifleaf/2` which selects its first argument (`skip`) if the input node does not have subnodes (nothing to be sorted), the second argument otherwise. Its definition is omitted. If there is at least one item, then `qsplit/1` is used to select one as a pivot element and arrange the rest into two groups according to their relation to the pivot element giving `A( gt( "BBCB") le( "AA"))`. The two groups are sorted (potentially in parallel) yielding `A( gt( "BBBC") le( "AA"))`, then `qmerge/0` is used to construct a node with subnodes containing the pivot element and the members of the two sorted groups in the appropriate order, so we get `sorted( "AAABBBC")`. Both `qsplit/1` and `qmerge/0` use a producer-consumer pair and enumeration to accomplish their task. Without these high level concepts and accompanying language features the program would hardly be understandable. The algorithm exhibits

both fine grained parallelism (by using `pc`) and coarse grained parallelism (by sorting two groups of items in parallel using `divide_conquer`). How much of this potential parallelism is realised, depends on the runtime scheduling of the independent threads.

Note also that the cost (complexity) of the algorithm is $O(n^2)$ on n items, the same as on the RAM.

6.5 Implementation

In this section we are going to describe a NOP simulator designed for conventional von Neumann machines. In fact it was written in C and runs under Linux. This implementation can immediately be transformed into a SM-MIMD parallel implementation. The memory of the simulator is partitioned into four *segments*, each for storing one of the following: *nodes*, *program*, *execution stacks*, *execution units*. The node segment is used to store node cells while the program is running. Management of this segment is very simple because all cells are of the same size, allocation request are served from the free list of node cells, deallocated cells are put back to the free list. The program is most conveniently represented as a tree whose internal nodes are labelled by first order NOP transformations (`seq`, `equ`, `all`), its leaves are labelled with primitive transformations. It may be necessary to introduce loops when recursive definitions are used (see `locate` above). An execution unit contains state information about a sequential thread of control. These are created when an `all` transformation is executed. One stack per execution unit is needed to handle `seq` transformations.

6.5.1 Data representation

Nodes are represented by a structure that contains four fields: *pointer to left neighbour*, *pointer to right neighbour*, *pointer to active subnode* and *value*. If the value of the node is itself a node, then the value field is a pointer to that node, otherwise it is an internal code identifying an atom. We have to be able to differentiate between these two cases, for example by using a single bit as tag.

6.5.2 Execution units

Execution units are structures containing the following four fields: *execution stack*, *pointer to node to be transformed*, *pointer to parent execution unit*, *number of active child execution units*. During processing, execution units are arranged into a tree. The root of this tree is the first execution unit created to execute the whole program. Descendants of an execution unit are created when an `all` transformation is executed. At any given instant, only the execution units at the leaves are active, the rest are passive waiting for their children to terminate. When an execution unit executes an `all` transformation, it creates child units, initialises them, sets its child count to be equal to their number and then becomes inactive. When an execution unit terminates, the child count of its parent is decremented and an active execution unit is selected for execution. When the parents' child count reaches zero, that parent is reactivated. Execution units are initialised by setting their node pointer to the node to be transformed and pushing the address of the transformation they have to execute onto their stack.

6.5.3 Cost of transformations

After learning about the internals of the simulator, we can turn our attention towards the complexity of transformations (see Table 6.1 and 6.2). Primitive transformations can be separated into two groups, members of one group have constant cost, these transformations require only a few pointer assignments to execute. The other group contains transformations whose cost depends on the size of the node tree they have to operate on. The size of a particular value tells us how many node cells are allocated for it and is defined by:

$$\begin{aligned} \text{size}(A) &= 0 & A \text{ is an atom} \\ \text{size}(V(N_1 \dots N_k)) &= 1 + \text{size}(V) + \text{size}(N_1) + \dots + \text{size}(N_k) & \text{for nodes} \end{aligned}$$

The cost of members of the second group reflects that they have to copy whole node trees from one place to another. The cost of compound transformations is a bit more difficult to compute because we have to take more factors into account. In Table 6.2, $C(X, N)$ denotes the cost of transforming node N with transformation X , the result naturally depends on the cost of the parameters and on how much extra work is needed to execute the transformation. `equ` has to compare two values, so its cost depends on the size of those values. `All` has to create as many execution units as there are subnodes, this is reflected in its cost. Finally, "cost" does not mean running time (especially in the case of `all`) on parallel computers, but amount of work to be done.

6.5.4 Multiprocessor implementation

The ultimate goal of the model is of course to allow efficient multiprocessor implementations. The simulator just discussed immediately translates into an SM-MIMD implementation. The only problem to solve is to regulate access to global data, that is critical sections of code have to be protected by using some mutual exclusion primitive (semaphores, for example). Critical sections are found in the node cell allocation/deallocation routines, in execution unit handling routines and depending on details, perhaps in stack handling procedures. It is also immediately obvious that there are ample opportunities to move data into private memory: each processor can be given a copy of the program, can have its own execution stack and some execution units can be stored privately. In fact, these possibilities are all present because the NOP model is extremely cache-friendly. Given a node cell, it is very simple to determine which cells are going to be used next (value and active subnode), given an execution unit, it is easy to find out which units come into play soon. Equally importantly, note that in the case of node cells, there is no cache coherence problem because there can only exist cells used exclusively by one execution unit. From these observations, we can conclude that the NOP model can be efficiently implemented on SM-MIMD architectures.

As we noted earlier, the DM-MIMD architecture is ideal regarding extensibility, so it would be advantageous to be able to implement the NOP model on it. This looks harder than the shared memory case, but there are better opportunities than with many other models and this is the main direction of research currently undertaken in connection with the NOP model. The problem with distributed memory is that communication costs can easily dominate the running time of programs if frequent and/or voluminous communication is necessary. In the case of NOP, its predictable data flow pattern helps us in determining exactly the data to send if communication becomes necessary, this way communication can be kept to an absolute minimum. Present plans contain dynamic load balancing techniques (see chapter 3) for work distribution among processors. Receiver initiated diffusion is the prime candidate to be used together with program optimisation methods.

6.5.5 Code listing guide

The implementation of the NOP abstract machine serves two purposes. First it is a proof of concept, it shows that the primitives can efficiently be implemented on existing computers. Secondly, it is a research tool that is used to explore the characteristics of the programs and to gain insight into future extensions to the model. The program listed in Appendix C accepts a NOP source program as its input, translates it into internal form, then executes it. It produces various lists in different phases, these can be used to find errors or evaluate the source programs.

All the source files are stored in a single directory. The Makefile (1) is used to coordinate the construction of the executable. The main function is found in `nop.c` (66) which simply initiates the successive phases of processing.

The components of the abstract machine were introduced earlier in this section. The representation of nodes is defined in `reprnode.h` (149) with the support functions in `reprnode.c` (180). The representation for transformations can be found in `reprtraf.h` (218) and `reprtraf.c` (291). Transformation are stored internally in a tree, loops can be introduced when the star notation is used in defining transformations.

Supplementary memory management functions are declared in `nopalloc.h` (609) and `mem.h` (638), with definitions in `nopalloc.c` (618) and `mem.c` (656).

Execution units and support functions are declared in `memexec.h` (713) and `memexec.c` (732), node memory management functions are declared in `memnode.h` (769) and `memnode.c` (779). Stacks are used by execution units, they are implemented in `memstack.h` (831) and `memstack.c` (848).

The syntax of the language the translator understands is essentially the same as the one used in this chapter, except that a '+' sign must precede every transformation declaration. The translator was implemented using the flex lexical analyser generator and the bison compiler generator software. These are by now standard language development tools in the Unix environment.

The lexical structure of the implemented NOP language is found in `nop.lex` (870) in flex format, with support functions in `lexparse.h` (1024). The syntax of the language is defined in `nop.y` (1052) in bison format. During the translation phase, symbol tables need to be built to store the transformation and atom names encountered in the source program. These tables are based on structures defined in `strtable.h` (1331), `strtable.c` (1352) and are implemented in `trafotab.h` (1472), `trafotab.c` (1480) and in `atomtab.h` (1533), `atomtab.c` (1541) respectively.

The functions in `noprav.h` (1609) and `noprav.c` (1633) are used to traverse node and transformation structures for diagnostic purposes.

The last modules belonging to the translator are `reprpars.h` (1914) and `reprpars.c` (2012) containing the data structures and functions used to build the parse tree in memory.

The executor source occupies a surprisingly small portion of the listing, but nevertheless this is the most important part where all the pieces come together. It is found in `nopexec.c` (2140).

Chapter 7

Summary

Framework Applications for Parallel Computers

Parallel computers and parallel programming have received considerable attention in recent years. This can be attributed to two factors. First, the ever growing need for processing power makes it necessary to look for new architectures because serial computers are approaching physical limits. It seems that the only way to increase the performance of computers is by using multiple processing elements which operate concurrently, in parallel. Second, the history of computing taught us that hardware and software are equally important, the complexity of problems can only be tackled by applying appropriate methods to design our systems. Parallelism may help allowing us to model real world and artificial phenomena in a natural way, thereby leading to cleaner, more reliable applications.

Besides its advantages, parallelism has drawbacks. The coordination of concurrent activities poses new problems and requires new methods, but these are inherently more difficult than in the sequential case. Further, despite its potential, parallel programming has not become widespread because there are no universally accepted general purpose parallel programming models. A large number of architectures and programming languages have been proposed, but at present, applications developed for a given architecture can only be ported to other architectures with substantial effort.

In the dissertation, I study several aspects of the parallel computing field, present applications that are the results of my work in this area and propose a parallel programming model (NOP) which can serve as a general programming model. The applications are not designed to solve a single problem, but provide a framework encapsulating functions that can be used to implement a range of systems similar in their characteristics.

In the introduction, hardware and software issues related to parallel computers are dis-

cussed. Two applications presented in the dissertation run on *transputer* networks, these are known as MIMD (Multiple Instruction stream/Multiple Data stream) machines according to the most popular classification scheme. This class is the most general, but at the same time the most difficult to handle. None of the architectures developed so far is ideal, this can partly be attributed to the lack of appropriate parallel programming models. The popular shared memory machines are easier to program, but they do not scale up very well. Distributed memory machines — better from an architectural point of view — are harder to program. Studying existing programming models, the NOP model is based on the idea that by restricting random access to memory, we can arrive at a model that is more successful in bridging the gap between hardware and software.

The central idea of parallel computation is that multiple processors work simultaneously to achieve a final result. For this to be possible, the problem to be solved has to be decomposed into independent parts: functional and/or domain decomposition can be used to achieve this. The database query language executor described in the dissertation uses functional decomposition, while data decomposition was used in the load balancing environment, the other application presented in the dissertation. Load balancing is the activity of ensuring that all processors perform useful work during the execution time of a given program. Some problems have a regular structure that allows us to define a decomposition in advance, before the program runs. In these cases load balancing is part of the design phase and is called *static load balancing*. There are several important problems that cannot be decomposed in advance, therefore load balancing becomes an integral part of the algorithm. This is called *dynamic load balancing*. A widely used dynamic load balancing method is the so-called processor farm, the particular load balancing method described in the dissertation can be regarded as a generalisation of the farm principle.

The first application is a parallel executor for a database query language called OFL (Object Functional Language). OFL is an experimental language designed to use with object oriented databases. The research in connection with OFL involved the specification and formal definition of the language. To assist the definition, an abstract machine has been developed, it can be implemented efficiently on the target architecture and is suitable to serve as basis for the formal definition of OFL. To execute OFL programs, a transputer-based database machine has been modelled. The database machine is modular, it clearly separates the responsibilities of the executor and that of its environment and defines the interface between them. The executor has been implemented on a transputer network and test results show that it is capable of speeding up OFL programs.

The other application is a load balancing test system that has been developed to provide an environment in which the means of mapping decomposable applications to distributed memory multi-processor systems can be explored. The processor farm approach of designating one master node to perform the task of controlling work distribution has been

replaced with a *processor commune* model in which work allocation is a community activity with each processing node combining master-worker functionality.

The test environment has three components: application model generator, application executor and load balancing analyser. The application model generator allows the computational characteristics of a particular program to be captured in the form of an application "mimic". This skeletal version, which provides a real time emulation of the application under consideration, is used as the basis for investigation of load balancing strategies with the application executor. This operates in the form of a local balancer process, installed on every processing element and works concurrently with the local application tasks. Because of the separation of load balancing and application functionality, the system can be used to explore the performance of different applications under a range of predefined work scheduling algorithms. The runtime data obtained are then passed to the analyser tools.

Results obtained using this toolset are presented: these show that good performance can be obtained under nearest neighbour scheduling approaches. The overheads associated with the employment of work allocation mechanism are low and the manner of task distribution ensures that the system operates with maximum available parallelism throughout the application execution. Future plans include the application of the test system to carry out a series of test which determine the boundaries of the effectiveness of the commune load balancing model.

Synthesized from work with the processor commune system, a building block has been defined which enables processes to run independently and communicate with each other without a predefined communication pattern. The processes work independently as long as they can, connection is established only when needed. The basic building block is successively extended in order to be able to deal with more complex situations. Any network of processes composed from the building block is deadlock-free.

Experience gained from the development of parallel systems, a general purpose parallel programming model (NOP) has been designed. It incorporates a number of novel features. It has a single data structure that makes random memory accesses unnecessary. This radical approach has both advantages and disadvantages, but it is argued that it is worth considering as a viable alternative method of memory management in the context of parallel processing. The model has both imperative and declarative features. Its simple imperative transformations can be easily and efficiently implemented, while larger programs can be constructed in a declarative style, benefiting from the substantial body of accumulated knowledge in the area of functional programming languages. The NOP transformations are commands in the imperative sense, they cause some action(s) to be performed. There are sequential, conditional and parallel composition operations. On the other hand, a NOP program is a mutually recursive equation set, in this sense it is declarative. These design choices are believed to produce a unique, radical, but satisfying model.

The ultimate goal of the model is of course to allow efficient multiprocessor implementations. A fully functional NOP simulator has been developed and implemented to carry out experiments and gain insight for further research. This sequential NOP simulator immediately translates into an SM-MIMD implementation, the only problem to solve is to regulate access to global data, but this can easily be done using standard mutual exclusion techniques. Research progresses toward investigating a combination of program transformations and dynamic load balancing to arrive at a distributed memory multiprocessor implementation.

Chapter 8

Összefoglaló (Summary in Hungarian)

Keretalkalmazások párhuzamos számítógépekre

A párhuzamos számítógépek és ezek programozása iránti érdeklődés az utóbbi években jelentősen megnövekedett. Ez két okra vezethető vissza. Először is, a mind nagyobb számítási kapacitást igénylő feladatok arra ösztönzik a kutatókat, hogy új utakat keressenek a számítógépek teljesítményének növelésére, mivel a hagyományos, szekvenciális gépek sebességének további növelése **fizikai korlátokba** ütközik. Úgy tűnik, a számítási kapacitás növelésének egyetlen módja több feldolgozóegység alkalmazása. Másrészt bizonyos esetekben az explicit párhuzamos konstrukciókat tartalmazó programozási nyelvek megfelelőbbek, mert így a program szerkezete jobban tükrözi a megoldandó probléma szerkezetét. A programtervezési elvek arra figyelmeztetnek, hogy a hardver és szoftver egyaránt fontos, így ez utóbbi szempont semmiképpen sem elhanyagolható.

A párhuzamosságnak előnyei mellett hátrányai is vannak. Az egyszerre zajló folyamatok koordinálása új problémákat vet fel, ezek megoldása természetüknél fogva nehezebb, mint a szekvenciális esetben. A párhuzamos számítógépek nem terjedtek el olyan mértékben, mint azt sokan jósolták. Ennek oka, hogy nem sikerült általános párhuzamos programozási modellt találni. Nagy számú architektúra és programozási nyelv született, de jelenleg egyik sem vált olyan mértékben elterjedtté, hogy hordozható alkalmazások készülhetnének.

A disszertációban a párhuzamos számítások több részterületét érintve olyan általános

alkalmazásokat mutatok be, amelyek nem egy adott probléma megoldására készültek, hanem keretet adnak speciálisabb alkalmazások elkészítéséhez. Ezen kívül javaslatot teszek egy olyan párhuzamos programozási modellre (NNode Processing model, NOP), amely igyekszik kiküszöbölni az előzőekben említett hiányosságokat.

A bevezetésben áttekintem a párhuzamos számítógépekkel kapcsolatos hardver és szoftver kérdéseket. A disszertációban bemutatott két alkalmazás *transputer* hálózaton fut, a transputer az architektúrák rendszerezésére leggyakrabban használt osztályozást alapul véve az ún. MIMD (Multiple Instruction stream/Multiple Data stream) osztályba tartozik. Ez az osztály a legáltalánosabb, de programozási szempontból legnehezebben kezelhető. A jelenleg ismert architektúrák egyike sem ideális, ez azonban részben a megfelelő párhuzamos számítási modellek hiányából is fakad. A közkedvelt közös memóriájú (Shared Memory) gépek könnyebben programozhatók, viszont nehezen bővíthetők. A felépítés szempontjából előnyösebb osztott memóriás (Distributed Memory) gépek viszont nehezebben programozhatók. Áttekintve a fontosabb programozási modelleket és alkalmazhatóságukat, a javasolt NOP modell azon a felismerésen alapul, hogy a közvetlen memóriáhozáférést korlátozva olyan párhuzamos modellt kaphatunk, amely sikeresebben tudja áthidalni a hardver és a szoftver közötti szakadékot.

A transputer hálózatokon futó — általánosabban az osztott memóriájú gépeken futó — alkalmazások esetén a legfontosabb megoldandó feladat az, hogy hogyan tudunk olyan részproblémákat elkülöníteni, amelyeket külön processzorok tudnak feldolgozni, felgyorsítva ezzel a program futását. A részproblémák elkülönítésére funkcionális vagy adat dekompozíció használható, a két bemutatott alkalmazás mindkettőre ad példát. Az elkülönített részproblémák feldolgozóegységekhez történő hozzárendelése egy másik olyan feladat, amelynek sikeres megoldásától nagyban függ a program futásideje. A hozzárendelés elvégezhető statikusan, azaz a program futtatása előtt, vagy dinamikusán, azaz a program futása közben. A két bemutatott alkalmazás megintcsak mindkettőre ad példát.

Az első keretalkalmazás egy adatbázislekérdező nyelv (OFL) párhuzamos kiértékelését teszi lehetővé. Az OFL egy kísérleti nyelv, amelyet objektumorientált adatbázisokhoz terveznek használni. A kutatás keretében korábbi résztervek és igényfelmérés alapján elkészült a nyelv specifikációja és formális definíciója. E definícióhoz megtervezésre került egy absztrakt gép, amely egyrészt hatékonyan implementálható a célarchitektúrán, másrészt alkalmas arra, hogy az OFL definíciójának alapjául szolgáljon. Az OFL programok végrehajtásához egy transzputerekből felépíthető adatbázis gépet modelleztünk. A gép moduláris szerkezetű, világosan elkülöníti egymástól a végrehajtó és a környezete feladatait és definiálja az együttműködésük szabályait. A tervek alapján

elkészült egy transzputerhálózaton futó implementáció és a teszteredmények azt mutatják, hogy a végrehajtó fel tudja gyorsítani a programokat.

A másik keretalkalmazás egy, a dinamikus terheléelosztás vizsgálatára alkalmas teszt rendszer. Számos olyan feladat van, amelynek sem algoritmus, sem adatai nem bonthatók fel független részekre a feldolgozás megkezdése előtt. Ezek hatékony párhuzamos megoldására dinamikus terheléelosztó algoritmusokat alkalmazhatunk, amelyek az adott problémát megoldó algoritmussal együtt futnak és látják el feladatukat.

Definiálásra került egy speciális feladatosztály, amely a deklaratív nyelveken írt programok végrehajtásának modelljén alapul, de általánosabban is használható. Elkészült egy teszt rendszer, amely három komponensből áll. Az alkalmazás generátor segítségével szintetikus alkalmazásokat generálhatunk, ezek futási idejű viselkedése megfelel a paraméterek által leírt valós alkalmazásénak. Az alkalmazás végrehajtó a szintetikus alkalmazást végrehajtja és közben adatokat gyűjt későbbi elemzés céljára. Az alkalmazás analízáló modul az előző kettő által szolgáltatott adatok alapján könnyen értelmezhető formában tálalja az eredményeket a felhasználónak. A teszt rendszer segítségével anélkül tudunk adatokat kapni egy tervezett alkalmazás viselkedéséről, hogy azt valójában elkészítettük volna.

Az alkalmazás végrehajtó az úgynevezett processzor kommuna elven működik, ez a jól ismert processzor farm modell általánosítása. A teszt rendszer segítségével méréseket végeztünk egy szintetikus logikai programokat kiértékelő alkalmazáson. Az eredmények azt mutatták, hogy jelentős teljesítménynövekedés érhető el a processzor kommuna modell használatával. Ezeket a később elkészült valós alkalmazás is megerősítette. A további tervek között szerepel a tesztrendszer felhasználása egy olyan tesztsorozat végrehajtására, amely a kommuna terheléelosztó algoritmus teljesítőképességének határait hivatott meghatározni.

Vizsgálat tárgyát képezte a terheléelosztó algoritmusban használt protokoll. Ennek helyessége nem nyilvánvaló, így szükségesnek látszott elemezni a viselkedését. Ennek eredményeképpen egy, a protokollok definiálásához felhasználható általános építőelemhez jutottunk. Az építőelemből összeállított folyamatgyűttek garantáltan holtpontmentesek.

A párhuzamos rendszerek fejlesztése során nyert tapasztalatokat felhasználva, egy általános párhuzamos számítási modell került kidolgozásra. A modell szakít azzal a realitásnak tűnő, de átgondolásra érett elvvel, hogy a programok adataikhoz közvetlen eléréssel férhetnek hozzá. A korlátlan adathozzáférést lecserélve egy jóval kötöttebb, csak implicit hozzáférést engedő sémával olyan modellhez jutunk, amelynek előnyös tulajdonságai figyelemre méltóak.

A modell alapja egy egyszerű hierarchikus adatstruktúra, amelynek csak kitüntetett elemeihez férhetnek hozzá közvetlenül a program utasításai (transzformációk). Az egyszerű transzformációk az alapvető műveleteket végzik el, míg a vezérlési szerkezetek összetett transzformációkkal valósíthatók meg. A közvetlen tárhozzáférés feladásának persze rövidtávú hátrányai is vannak, ezek azonban a megfelelő programozástechnikai módszerek kidolgozásával kiküszöbölhetőnek látszanak. Elkészült az absztrakt gép egy C nyelven írt implementációja, amelynek segítségével további tapasztalatok nyerhetők. Ez az implementáció közvetlenül átalakítható egy közös memóriát használó többprocesszoros gépen futó alkalmazássá. További kutatás szükséges annak megállapítására, hogy elosztott memóriájú gépeken milyen implementációs stratégia a legmegfelelőbb.

Bibliography

[papers marked with ▷ appeared in periodicals listed in Zentralblatt für Mathematik]

- [1] L. Schrettner, I. E. Jelly, Z. Alexin, T. Gyimóthy: **Parallel Execution of OFL Programs**, *Large Parallel Databases Technical Report*, Copernicus Project CP 93:6638, 1995
- ▷ [2] L. Schrettner, T. Gyimóthy, Z. Alexin, J. Toczki: **Parallel Execution of Object Functional Queries**, *Annales Univ. Sci. Budapestinensis, Sectio Computatorica*, Vol. 17, 1998, pp. 339–354
- [3] L. Schrettner, J. Toczki: **Dynamic Load Balancing for Decomposable Problems**, *Parallel Processing in Education*, Impact TEMPUS JEP's and Hungarian Transputer Users Group's Workshop, March 1993
- [4] L. Schrettner, I. E. Jelly: **Investigation of Dynamic Load Balancing in Distributed Memory Multiprocessor Machines**, *Proc. of 15th Intl. Conf. on Information Technology Interfaces*, Pula, Croatia, June 1993
- ▷ [5] L. Schrettner, I. E. Jelly: **A Test Environment for Investigation of Dynamic Load Balancing in Transputer Networks**, *Transputer Applications and Systems '93*, Transputer and Occam Engineering Series, Vol. 36, IOS Press, 1993, pp. 284–295
- ▷ [6] J. Toczki, L. Schrettner: **Attribute Grammar Applications**, *Annales Univ. Sci. Budapestinensis, Sectio Computatorica*, Vol. 17, 1998, pp. 367–380
- [7] L. Schrettner, M. Bohus: **A Parallel Design Method for Producing Valid Protocols**, *Proc. of 8th Symposium on Microcomputer and Microprocessor Applications*, October 1994, pp. 689–697
- ▷ [8] L. Schrettner, I. E. Jelly: **Dynamic Process Interaction**, *Parallel Programming and Java*, Concurrent Systems Engineering Series, Vol. 50, IOS Press, 1997, pp. 261–273
- [9] L. Schrettner: **The NOP Parallel Programming Model**, submitted for publication
- [10] A. S. Tannenbaum: **Parallel Architectures**, in *Structured Computer Organization*, 3rd Edition, Ch. 8.2, Prentice-Hall, 1990
- [11] M. J. Flynn: **Some Computer Organizations and their Effectiveness**, *IEEE Trans. Computers*, Vol. C-21, September 1972, pp. 948–960
- [12] B. P. Lester: **The Art of Parallel Programming**, Ch. 3, Prentice-Hall, 1993
- [13] R. Cypher, J. Sanz: **The SIMD Model of Parallel Computation**, Springer-Verlag, 1994
- [14] D. Sima, T. Fountain, P. Kacsuk: **Advanced Computer Architectures. A Design Space Approach**, Ch. 3, Addison Wesley, 1997

- [15] INMOS Ltd.: **Transputer Databook**, 2nd Edition, 1989
- [16] INMOS Ltd.: **The Transputer Development and iq Systems Databook**, 2nd Edition, 1991
- [17] INMOS Ltd.: **Transputer Instruction Set**, Prentice Hall, 1988
- [18] M. D. May, P. W. Thompson, P. H. Welch (Eds): **Networks, Routers and Transputers: Function, Performance and Application**, Transputer and OCCAM Engineering Series, IOS Press, 1993
- [19] P. Kacsuk, Sz. Ferenczi: **Parallel and Concurrent Programming in Multitransputer Systems** BME Mérnöktoábbképző Intézet, Budapest, 1993 (in Hungarian)
- [20] Telmat Informatique: **T.Node Hardware Manual**
- [21] Telmat Informatique: **T.Node Software Manual**
- [22] I. E. Jelly, I. Gorton: **Software Engineering for Parallel Systems**, *Information and Software Technology Journal*, Vol. 36, No. 7, 1994, pp. 381–396
- [23] I. E. Jelly, I. Gorton, P. Croll (Eds): **Software Engineering for Parallel and Distributed Systems**, *Proc. 1st IFIP Intl. Workshop on Parallel and Distributed Software Engineering*, March 1996
- [24] I. E. Jelly: **A Parallel Process Model and Architecture for a Pure Logic Language**, PhD Thesis, Sheffield City Polytechnic, 1990
- [25] J. P. Gray, I. E. Jelly: **A Hybrid Transputer Based Architecture for Parallel Logic Language Execution**, *Proc. of First World Transputer Conference*, Transputing-91, IOS Press, April 1991
- [26] S. C. Winter, P. Kacsuk: **Software Engineering for Parallel Processing**, *Proc. of 8th Symposium on Microcomputer and Microprocessor Applications*, October 1994
- [27] L. G. Valiant: **A Bridging Model for Parallel Computation**, *Comm. ACM*, Vol. 33, No. 8, August 1990, pp. 103–111
- [28] B. M. Maggs, L. R. Matheson, R. E. Tarjan: **Models of Parallel Computation: A Survey and Synthesis**, *Proc. of 28th Hawaii Intl. Conf. On System Sci. (HICSS)*, Vol. 2, 1995, pp. 61–70
- [29] G. Michaelson: **An Introduction to Functional Programming Through Lambda Calculus**, International Computer Science Series, Addison-Wesley, 1988
- [30] Å. Wikström: **Functional Programming Using Standard ML**, Prentice Hall International Series in Computer Science, Prentice Hall, 1987
- [31] P. Hudak: **Conception, Evolution, and Application of Functional Programming Languages**, *ACM Computing Surveys*, Vol. 21, No. 3, ACM Press, September 1989, pp. 359–411
- [32] W. Clocksin, C. Mellish: **Programming in PROLOG**, Springer Verlag, 1981
- [33] S. A. Cook, R. A. Reckhow: **Time Bounded Random Access Machines**, *J. Comput. Systems Sci.*, Vol. 7, 1973, pp. 354–375

- [34] R. M. Karp, V. Ramachandran: **Parallel Algorithms for Shared-Memory Machines**, in *Handbook of Theoretical Computer Science*, Vol. A, Ch. 17, Elsevier, 1990, pp. 869–942
- [35] C. A. R. Hoare: **Monitors: An Operating System Structuring Concept**, *Comm. ACM*, Vol. 17, October 1974, pp. 549–557
- [36] E. Bal, J. G. Steiner, A. S. Tannenbaum: **Programming Languages for Distributed Computing Systems**, *ACM Computing Surveys*, Vol. 21, No. 3, ACM Press, September 1989, 261–322
- [37] V. S. Sunderan: **PVM: A Framework for Distributed Computing**, *Concurrency, Practice and Experience*, Vol. 2, No. 4, December 1990, pp. 315–339
- [38] P. B. Hansen: **Distributed Processes: A Concurrent Programming Concept**, *Comm. ACM*, Vol. 21, No. 11, November 1978, pp. 934–941
- [39] C. A. R. Hoare: **Communicating Sequential Processes**, Prentice Hall, 1985
- [40] A. W. Roscoe: **The Theory and Practice of Concurrency**, Prentice Hall, 1998
- [41] D. Gelernter, N. Carriero: **Applications Experience with Linda**, in *Proc. PPEALS 1998*, SIGPLAN Not. (ACM), Vol. 23, No. 9, September 1988, pp. 173–187
- [42] B. K. Szymanski (Ed): **Parallel Functional Languages and Compilers**, ACM Press Frontier Series, ACM Press, 1991
- [43] P. Kelly: **Functional Programming for Loosely-coupled Multiprocessors**, Research Monographs in Parallel and Distributed Computing, Pitman, 1989
- [44] E. Shapiro: **The Family of Concurrent Logic Programming Languages**, *ACM Computing Surveys*, Vol. 21, No. 3, ACM Press, September 1989, pp. 412–510
- [45] INMOS Ltd.: **OCCAM 2 Reference Manual**, Prentice-Hall, 1988
- [46] INMOS Ltd.: **OCCAM 2 Toolset User Manual**, *IMS D00205-DOCA*, 1991
- [47] G. Jones, M. Goldsmith: **Programming in OCCAM 2**, Prentice-Hall, 1988
- [48] INMOS Ltd.: **ANSI C Toolset User Manual**, *IMS D0214-DOCA*, 1990
- [49] W. Day: **Farming: Towards a Rigorous Definition and Efficient Transputer Implementation**, *Transputer Systems—Ongoing Research*, IOS Press, 1992
- [50] L. A. Cowl: **How to Measure, Present and Compare Parallel Performance**, *IEEE Parallel and Distributed Technology*, Spring 1994, pp. 9–25
- [51] R. G. G. Cattell (Ed): **Object Databases: The ODMG Standard**, Morgan Kaufman, 1993
- [52] G. Gardarin, F. Machuca, P. Pucheral: **A Functional Execution Model to Evaluate Object-Oriented Queries**, *PRISM Technical Report*, University of Versailles, France, 1994
- [53] G. Gardarin et al: **OFL: An Object Functional Language to Map Extended SQL**, *Large Parallel Databases Technical Report*, Copernicus Project CP 93:6638, 1994

- [54] **PROF-LP User's Guide**, Research Group on the Theory of Automata, Szeged, 1987
- [55] H. Alblas: **Attribute Evaluation Methods**, *Memoranda Informatica 89-20*, Faculteit der Informatica, University Twente, the Netherlands, 1989
- [56] J. Toczki: **Attribute Grammars and their Applications**, *Dr. Univ. thesis*, Departments of Informatics, József Attila University, Szeged, Hungary, 1991, (in Hungarian)
- [57] L. Schrettner: **Dynamic Attribute Evaluation**, *MSc. Thesis*, Dept. of Computer Science, JAU, 1992, (in Hungarian)
- [58] M. Tóth: **Parallel execution of logic programs**, *BSc. Thesis*, Dept. of Computer Science, JAU, 1999, (in Hungarian)
- [59] F. W. Burton, M. M. Huntbach: **Virtual Tree Machines**, *IEEE Trans. on Computers*, Vol. C-33, No. 3, March 1984
- [60] D. L. McBurney, M. R. Sleep: **Transputer-Based Experiments with the ZAPP Architecture**, *University of East Anglia report no. SYS-C86-10*, October 1986
- [61] F. C. H. Lin, R. M. Keller: **The Gradient Model Load Balancing Method**, *IEEE Trans. on Software Engineering*, Vol. SE-13, No. 1, January 1987
- [62] G. Cybenko: **Dynamic Load Balancing for Distributed Memory Multiprocessors**, *J. Parallel and Distributed Computing*, Vol. 7, October 1989
- [63] M. H. Willebeck-Lemair, A. P. Reeves: **Strategies for Dynamic Load Balancing on Highly Parallel Computers**, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 9, September 1993
- [64] P. K. K. Loh, W. J. Hsu, C. Wentorg, N. Sriskanthan: **How Network Topology Affects Dynamic Load Balancing**, *IEEE Parallel and Distributed Technology*, Fall 1996, pp. 25-35
- [65] G. J. Holzmann: **Design and Validation of Computer Protocols**, Prentice-Hall International, 1991
- [66] P. Welch, G. Justo and C. Willcock: **High-Level Paradigms for Deadlock-Free High-Performance Systems**, *Transputer Applications and Systems '93*, IOS Press, 1993 pp. 981-1004
- [67] J. Misra: **Detecting Termination of Distributed Computation Using Markers**, *Proc. of the 2nd annual ACM Symposium on Principles of DC*, Montreal, August 1983, pp. 290-294
- [68] M. Raynal: **Distributed Algorithms and Protocols**, Wiley, 1988

Appendix A

OFL executor code listing

makefile

```
1
2 #
3 # Makefile for OFL (Object Functional Language) that
4 #
5 # 1. generates .c files from the attribute grammar specifications
6 # 2. compiles the generated code into an executable program
7 #
8
9 GENLIB = ..\clp
10 PROJECT = ofl
11
12 all: $(PROJECT).lex.c $(PROJECT).c $(PROJECT).exe
13 #all: $(PROJECT).exe
14
15 $(PROJECT).sym: $(PROJECT).hlp $(PROJECT).lxd
16 -2 $(GENLIB)\cagpars $(PROJECT);
17
18 $(PROJECT).tab: $(PROJECT).sym $(PROJECT).syn
19 -2 $(GENLIB)\ullitest $(PROJECT);
20
21 $(PROJECT).vis: $(PROJECT).sym $(PROJECT).syn
22 -2 $(GENLIB)\ulatest $(PROJECT);
23
24
25 $(PROJECT).lex.c: $(PROJECT).sym
26 -2 $(GENLIB)\clexsem $(PROJECT),,$(PROJECT).lex;
27
28 $(PROJECT).sms: $(PROJECT).tab $(PROJECT).vis
29 $(GENLIB)\usorthlp $(PROJECT),,$(PROJECT);
30
31 $(PROJECT).c: $(PROJECT).sms
32 $(GENLIB)\csortgen $(PROJECT),,$(PROJECT);
33
34 #
35 # Here comes the compiling of the C/C++ source
36 #
37
38 CC = cl
39 LINK = link
40
41 CFLAGS = -AL -c -Zid -Odr
42
43 c_agstd.obj: c_agstd.c
44 $(CC) $(CFLAGS) c_agstd.c
45
```

```
46
47 c_usr.obj: c_usr.c
48 $(CC) $(CFLAGS) c_usr.c
49
50 $(PROJECT).obj: $(PROJECT).c $(PROJECT).parse.h
51 $(CC) $(CFLAGS) $(PROJECT).c
52
53 $(PROJECT).lex.obj: $(PROJECT).lex.c
54 $(CC) $(CFLAGS) $(PROJECT).lex.c
55
56 $(PROJECT).parse.obj: $(PROJECT).parse.c $(PROJECT).parse.h
57 $(CC) $(CFLAGS) $(PROJECT).parse.c
58
59 $(PROJECT).x.obj: $(PROJECT).x.c $(PROJECT).parse.h
60 $(CC) $(CFLAGS) $(PROJECT).x.c
61
62 $(PROJECT).exe: $(PROJECT).obj $(PROJECT).lex.obj $(PROJECT).parse.obj\
63 $(PROJECT).x.obj c_agstd.obj c_usr.obj
64 $(LINK) /NOD/MAP/STACK:32000 $(PROJECT) $(PROJECT).lex\
65 $(PROJECT).parse $(PROJECT).x c_usr c_agstd,,llibce /CO;
66
```

std.h

```
67 #ifndef std_included
68 #define std_included
69
70 typedef unsigned int uint;
71
72 typedef enum {
73 NO=0, False=0,
74 YES=1, True=1
75 } bool;
76
77 typedef char *string;
78
79 #endif
```

A.1 Parser

ofl.lxd

80 lexical description ofllex


```

81 character sets
82 Quote      = ",";
83 AnyStr      = Any - Quote;
84 CommentBegin = "{";
85 CommentEnd   = "}";
86 CommentChar = Any - CommentEnd;
87 Escape      = "\";
88 NonZero     = Digit - "0";
89 MixLetter   = UpperCase / LowerCase;
90 NameChar    = MixLetter / Digit / "_";
91
92 token classes
93 Funcname    = UpperCase NameChar * ;
94 Name        = NameChar*;
95 Quoted      = Quote ( 'Escape Quote' / ( Escape Escape ) / AnyStr)* Quote;
96 Specials    = Default Tokens;
97 Spaces      = ( Space + ) / EndLine ;
98 Comment     = CommentBegin CommentChar* CommentEnd ;
99
100 begin
101 Funcname    -> Functionname / KeyStrings;
102 Name        -> Identifier / KeyStrings;
103 Quoted      -> Identifier / KeyStrings;
104 Specials    -> KeyStrings;
105 Spaces      -> ;
106 Comment     -> ;
107
108 end ;
109
110 end of lexical description

```

ofl.hlp

```

112 attribute grammar ofl
113
114 synthesized attributes are
115 pt:      ParseTreePtr;
116 ptl:     PTLListPtr;
117 arity:   uint;
118
119 unit ofl;
120
121 include : 'c_usr.h', 'oflparse.h' ;
122
123 nonterminals are
124
125 ofl,
126 function, predicate,
127 builtin, external,
128 iftrue, iffals,
129 argument
130
131 arguments,
132 argumentlist,
133 argumentlistrem
134
135 objectref
136
137 unit ofllex;
138
139 tokens are
140 Identifier has Val, Str;
141 Functionname has Val, Str;
142
143 terminals are
144
145 "(", ")", " ",
146 "assign", "while", "sequence", "forall", "forany", "if",
147 "Current", "First", "Next";
148
149 productions are
150
151 ofl = function;
152 do

```

```

153 pt <- ofl_close( pt, function.pt);
154 end;
155
156 function = external;
157 function = builtin;
158
159 external = "Current" "(" objectref ")";
160 do
161 pt <-
162 AddPTNode( pt, NATTR_EXTERNAL_CURRENT, " Current", 1, objectref.pt);
163 end;
164
165 external = "First" "(" argument ")";
166 do
167 pt <-
168 AddPTNode( pt, NATTR_EXTERNAL_FIRST, " First", 1, argument.pt);
169 end;
170
171 external = "Next" "(" argument ")";
172 do
173 pt <-
174 AddPTNode( pt, NATTR_EXTERNAL_NEXT, " Next", 1, argument.pt);
175 end;
176
177 external = Functionname "(" arguments ")";
178 do
179 pt <-
180 AddPTNodeList( pt, NATTR_EXTERNAL_OTHER,
181 Functionname.Str, arguments.arity, arguments.ptl);
182 end;
183
184 arguments = ;
185 do
186 ptl = NULL;
187 arity = 0;
188 end;
189
190 arguments = argumentlist;
191
192 argumentlist = argument argumentlistrem;
193 do
194 arity = argumentlistrem.arity+1;
195 ptl <-
196 AddParam( ptl, argument.pt, argumentlistrem.ptl);
197 end;
198
199 argumentlistrem= ;
200 do
201 ptl = NULL;
202 arity = 0;
203 end;
204
205 argumentlistrem= "," argumentlist;
206
207 argument = function;
208
209 argument = objectref;
210
211 objectref = Identifier;
212 do
213 pt <-
214 AddPTNode( pt, NATTR_OBJECT_REFERENCE, Identifier.Str, 0, NULL);
215 end;
216
217 builtin = "assign" "(" objectref "," argument ")";
218 do
219 pt <-
220 AddPTNode( pt, NATTR_INTERNAL_ASSIGN,
221 " assign", 2, objectref.pt, argument.pt);
222 end;
223
224 builtin = "sequence" "(" arguments ")";
225 do
226 pt <-
227 AddPTNodeList( pt, NATTR_INTERNAL_SEQUENCE,

```


A.1. PARSER

```

228      "sequence", arguments.arity, arguments.pt1);
229  end;
230
231 builtin      = "while"      (" predicate ", " function ");
232 do
233   pt <-
234   AddPTNode( pt, NATTR_INTERNAL_WHILE,
235             " while", 2, predicate.pt, function.pt);
236 end;
237
238 builtin      = "if"         (" predicate ", " iftrue", " iffalse");
239 do
240   pt <-
241   AddPTNode( pt, NATTR_INTERNAL_IF,
242             " if", 3, predicate.pt, iftrue.pt, iffalse.pt);
243 end;
244
245 iftrue       = argument;
246
247 iffalse      = argument;
248
249 builtin      = "forany"     (" objectref ", " predicate ", " function ");
250 do
251   pt <-
252   AddPTNode( pt, NATTR_INTERNAL_FORANY,
253             " forany", 3, objectref.pt, predicate.pt, function.pt);
254 end;
255
256 builtin      = "forall"     (" objectref ", " predicate ", " function ");
257 do
258   pt <-
259   AddPTNode( pt, NATTR_INTERNAL_FORALL,
260             " forall", 3, objectref.pt, predicate.pt, function.pt);
261 end;
262
263 predicate    = argument;
264
265 end of attribute grammar

```

oflparse.h

```

267 /*****
268 #ifndef parse_included
269 #define parse_included
270
271 #include "std.h"
272
273 typedef struct ParseTreeNode  *ParseTreePtr;
274
275 typedef struct ParseTreeNode {
276   uint      NodeAttrib;
277   string    NodeLabel;
278   uint      NofParams;
279   void      *info;          // NULL in parse tree
280   ParseTreePtr ( *Param)[];
281 } ParseTreeNode;
282
283 typedef struct PTLItem        *PTListPtr;
284
285 typedef struct PTLItem {
286   PTLItemPtr  next;
287   ParseTreePtr pt;
288 } PTLItem;
289
290 /*****
291 // values for NodeAttrib
292
293 // external functions
294 #define NATTR_EXTERNAL_CURRENT  0
295 #define NATTR_EXTERNAL_FIRST    1
296 #define NATTR_EXTERNAL_NEXT     2
297 #define NATTR_EXTERNAL_OTHER    3
298
299 // internal functions

```

```

300 #define NATTR_INTERNAL_ASSIGN    4
301 #define NATTR_INTERNAL_SEQUENCE 5
302 #define NATTR_INTERNAL_WHILE    6
303 #define NATTR_INTERNAL_IF       7
304 #define NATTR_INTERNAL_FORANY   8
305 #define NATTR_INTERNAL_FORALL   9
306 #define NATTR_INTERNAL_FORALL_X 10 // not used initially
307
308 // object references
309 #define NATTR_OBJECT_REFERENCE 11
310 #define NATTR_OBJREF_ASSIGNED 12 // not used initially
311 #define NATTR_OBJREF_FORALL_X 13 // not used initially
312 /*****
313
314 extern ParseTreePtr  ParseTree;
315
316 void oflx( void);
317
318 #endif

```

oflparse.c

```

319 /*****
320 #include <stdio.h>
321 #include <stdlib.h>
322 #include <string.h>
323 #include "std.h"
324 #include "oflparse.h"
325
326 ParseTreePtr  ParseTree;
327
328 /*****
329 void CheckPtr( void *p) {
330   if ( p == NULL) {
331     fputs( "out of memory", stderr);
332     abort();
333   }
334 }
335
336 /*****
337 static void ProcessString( string str) {
338   uint i, j, l;
339
340   if ( str[ 0] == '\') {
341     l = strlen( str);
342     for ( i=0, j=1; j < l-1; i++, j++) {
343       if ( str[ j] == '\')
344         j++;
345       str[ i] = str[ j];
346     }
347     str[ i] = '\0';
348   }
349 }
350
351 /*****
352 static void AllocNode( ParseTreePtr *ptp, uint attrib, string label, uint arity) {
353   ParseTreePtr pt;
354
355   pt = ( ParseTreePtr) malloc( sizeof ( ParseTreeNode));
356   CheckPtr( pt);
357
358   pt->NodeAttrib = attrib;
359
360   pt->NodeLabel = ( string) malloc( strlen( label));
361   CheckPtr( pt->NodeLabel);
362
363   strcpy( pt->NodeLabel, label+1);
364   ProcessString( pt->NodeLabel);
365
366   pt->NofParams = arity;
367
368   pt->info = NULL;
369
370   pt->Param = ( ParseTreePtr (*)[]) malloc( arity * sizeof ( ParseTreePtr));
371   CheckPtr( pt->Param);

```



```

372 *ptp = pt;
373 }
374
375 /*****
376 void AddPTNode( ParseTreePtr *ptp, uint attrib, string label,
377               uint nofparm,
378               ParseTreePtr pt1, ParseTreePtr pt2, ParseTreePtr pt3) {
379
380     AllocNode( ptp, attrib, label, nofparm);
381
382     if ( nofparm >= 1)
383         (*( *ptp)->Param)[ 0] = pt1;
384     if ( nofparm >= 2)
385         (*( *ptp)->Param)[ 1] = pt2;
386     if ( nofparm >= 3)
387         (*( *ptp)->Param)[ 2] = pt3;
388 }
389
390 /*****
391 void AddPTNodeList( ParseTreePtr *ptp, uint attrib, string label,
392                   uint arity,
393                   PTLlistPtr list) {
394
395     uint i;
396
397     AllocNode( ptp, attrib, label, arity);
398
399     for ( i = 0; i < arity; i++) {
400         (*( *ptp)->Param)[ i] = list->pt;
401         list = list->next;
402     }
403 }
404
405 /*****
406 void AddParam( PTLlistPtr *ptl, ParseTreePtr pt, PTLlistPtr rptl) {
407
408     *ptl = ( PTLlistPtr) malloc( sizeof ( PTLlistItem));
409     CheckPtr( *ptl);
410
411     ( *ptl)->next = rptl;
412     ( *ptl)->pt = pt;
413 }
414
415 /*****
416 void ofl_close( ParseTreePtr *pt, ParseTreePtr fpt) {
417     uint i;
418
419     extern int xxErrFlag;
420
421     ParseTree = *pt = fpt;
422     if ( ! xxErrFlag)
423         oflx();
424 }
425
426 /*****
427

```

A.2 OFL Abstract Machine Code Generator

```
ginstr.h
```

```

428 // token values
429 #define INSTR_TOKEN_JUMP      0
430 #define INSTR_TOKEN_JTRUE    1
431 #define INSTR_TOKEN_JFALSE   2
432 #define INSTR_TOKEN_LOAD     3
433 #define INSTR_TOKEN_CALL     4
434 #define INSTR_TOKEN_ASSIGN   5
435 #define INSTR_TOKEN_SEND     6
436 #define INSTR_TOKEN_OPER     7
437

```

```

438 #define jumpi( tokencode)      ( tokencode <= INSTR_TOKEN_JFALSE)
439
440 static char *tokenstrings[] = {
441     "jump",      /* INSTR_TOKEN_JUMP */
442     "jtrue",     /* INSTR_TOKEN_JTRUE */
443     "jfalse",    /* INSTR_TOKEN_JFALSE */
444     "load",      /* INSTR_TOKEN_LOAD */
445     "call",      /* INSTR_TOKEN_CALL */
446     "assign",    /* INSTR_TOKEN_ASSIGN */
447     "send",      /* INSTR_TOKEN_SEND */
448     "oper",      /* INSTR_TOKEN_OPER */
449 };
450
451 /* oper arguments */
452 #define OPER_ARG_MARK      0
453 #define OPER_ARG_CLEAR    1
454 #define OPER_ARG_POP      2
455
456 static char *operargstrings[] = {
457     "mark",      /* OPER_ARG_MARK */
458     "clear",     /* OPER_ARG_CLEAR */
459     "pop",       /* OPER_ARG_POP */
460 };
461

```

```
oflx.c
```

```

462 /*****
463 #include <stdio.h>
464 #include <stdlib.h>
465 #include <string.h>
466 #include "oflparse.h"
467
468 static FILE *of;          // object file variable
469
470 typedef struct tableentry {
471     uint   attr;
472     string id;
473 } tableentry;
474
475 typedef struct {
476     tableentry * const tableptr;
477     const uint   tablesize;
478     uint         nof_entries;
479 } table_desc;
480
481 /*
482 * direct object ids
483 */
484 #define DID_TABLE_SIZE 100
485 static tableentry did_table[ DID_TABLE_SIZE] = {
486     { 0, "false" },
487     { 0, "true" },
488     { 0, "nil" }    // predefined ids
489 };
490
491 static table_desc did_table_desc = {
492     did_table, DID_TABLE_SIZE, 3
493 };
494
495 /*
496 * indirect object ids
497 */
498 #define IID_TABLE_SIZE 100
499 static tableentry iid_table[ IID_TABLE_SIZE];
500
501 static table_desc iid_table_desc = {
502     iid_table, IID_TABLE_SIZE, 0
503 };
504
505 /*
506 * external function names
507 */
508 #define FNID_TABLE_SIZE 100
509 static tableentry fnid_table[ FNID_TABLE_SIZE] = {

```


A.2. OFL ABSTRACT MACHINE CODE GENERATOR

```

510 { NATTR_EXTERNAL_CURRENT, "Current"},
511 { NATTR_EXTERNAL_FIRST, "First" },
512 { NATTR_EXTERNAL_NEXT, "Next" }, // mandatory functions
513 };
514
515 static table_desc fnid_table_desc = {
516     fnid_table, FNID_TABLE_SIZE, 3
517 };
518
519 static uint ind_table_index; // set in JoinTables, first assigned id
520
521 /*****
522 typedef struct instruction *iptr;
523
524 typedef struct instruction {
525     uint token;
526     int arg;
527     string rem;
528 } instruction;
529
530 #include "ginstr.h"
531
532 typedef struct block *bptr;
533
534 typedef struct blockentry {
535     enum { e_block, e_instr} bi;
536     union {
537         iptr i;
538         bptr b;
539     };
540 } blockentry;
541
542 typedef struct block {
543     uint blength;
544     uint tlength;
545     blockentry bvector[];
546 } block;
547
548 #define CODEVEC_SIZE 100
549
550 static bptr code[CODEVEC_SIZE];
551 static uint codeidx;
552
553 /*****
554 bool Lookup( table_desc *td, string str, uint *idx, uint *attr) {
555     uint i;
556     for ( i = 0; i < td->nof_entries; i++)
557         if ( strcmp( td->tableptr[ i].id, str) == 0) {
558             *idx = i;
559             *attr = td->tableptr[ i].attr;
560             return YES;
561         }
562     // *idx, *attr not modified
563     return NO;
564 }
565
566 /*****
567 uint Insert( table_desc *td, string str, uint attr) {
568     uint idx, oattr;
569     if ( Lookup( td, str, &idx, &oattr)) {
570         td->tableptr[ idx].attr = attr;
571         return idx;
572     }
573     if ( td->nof_entries < td->tablesize) {
574         td->tableptr[ td->nof_entries].id = str; // does not allocate space !!!!
575         td->tableptr[ td->nof_entries].attr = attr;
576         return td->nof_entries++;
577     }
578     else {
579         fputs( "table full.", stderr);
580         abort();
581     }
582 }
583
584 */

```

```

585 }
586 }
587
588 /*****
589 bool Backbone( ParseTreePtr ptp) {
590     if ( ptp->NodeAttrib == NATTR_INTERNAL_FORALL) {
591         ptp->NodeAttrib = NATTR_INTERNAL_FORALL_X;
592         Backbone( ( *ptp->Param)[ 2]);
593         return YES;
594     }
595     else {
596         int i;
597         for ( i = ptp->NofParams-1; i >= 0; i--)
598             if ( Backbone( ( *ptp->Param)[ i]))
599                 return YES;
600         return NO;
601     }
602 }
603
604 /*****
605 void FillTables( ParseTreePtr ptp) {
606     ParseTreePtr p;
607     uint idx, attr;
608     uint i, start;
609
610     switch ( ptp->NodeAttrib) {
611         case NATTR_INTERNAL_ASSIGN: {
612             p = ( *ptp->Param)[ 0];
613             if ( Lookup( &iid_table_desc, p->NodeLabel, &idx, &attr)) {
614                 fprintf( stderr, "%s: identifier used before assign.", p->NodeLabel);
615                 abort();
616             }
617             Insert( &iid_table_desc, p->NodeLabel, NATTR_OBJREF_ASSIGNED);
618             start = 1;
619             break;
620         }
621         case NATTR_INTERNAL_FORALL_X: {
622             p = ( *ptp->Param)[ 0];
623             attr = NATTR_OBJECT_REFERENCE;
624             Lookup( &iid_table_desc, p->NodeLabel, &idx, &attr);
625             Insert( &iid_table_desc, p->NodeLabel, NATTR_OBJREF_FORALL_X);
626             start = 1;
627             break;
628         }
629         case NATTR_EXTERNAL_CURRENT: {
630             p = ( *ptp->Param)[ 0];
631             start = 0;
632             if ( Lookup( &iid_table_desc, p->NodeLabel, &idx, &attr))
633                 if ( attr == NATTR_OBJREF_FORALL_X) {
634                     p->NodeAttrib = NATTR_OBJREF_FORALL_X;
635                     start = 1;
636                 }
637             break;
638         }
639         case NATTR_EXTERNAL_OTHER: {
640             Insert( &fnid_table_desc, ptp->NodeLabel, NATTR_EXTERNAL_OTHER);
641             start = 0;
642             break;
643         }
644         case NATTR_OBJECT_REFERENCE: {
645             if ( !Lookup( &iid_table_desc, ptp->NodeLabel, &idx, &attr))
646                 Insert( &iid_table_desc, ptp->NodeLabel, NATTR_OBJECT_REFERENCE);
647             start = 0;
648             break;
649         }
650         default: {
651             start = 0;
652             break;
653         }
654     }
655 }
656
657 for ( i = start; i < ptp->NofParams; i++)
658     FillTables( ( *ptp->Param)[ i]);
659
660 */

```



```

660 if ( ptp->NodeAttrib == NATTR_INTERNAL_FORALL_X)
661     Insert( &iid_table_desc, p->NodeLabel, attr);
662 }
663
664 /*****
665 void JoinTables( void) {
666     uint i;
667
668     if ( did_table_desc.nof_entries+iid_table_desc.nof_entries >
669         did_table_desc.tablesize) {
670         fprintf( stderr, "cannot join tables.");
671         abort();
672     }
673 }
674
675 ind_table_index = did_table_desc.nof_entries;
676
677 for ( i = 0; i < iid_table_desc.nof_entries; i++)
678     did_table[ i+did_table_desc.nof_entries] = iid_table[ i];
679
680 did_table_desc.nof_entries += iid_table_desc.nof_entries;
681 }
682 /*****
683 iptr AllocInstr( uint token, int arg, string remark) {
684     iptr instr;
685
686     instr = ( iptr) malloc( sizeof ( instruction));
687     CheckPtr( instr);
688     instr->token = token;
689     instr->arg = arg;
690     instr->rem = NULL;
691     if ( remark != NULL) {
692         instr->rem = ( string) malloc( strlen( remark)+1);
693         CheckPtr( instr->rem);
694         strcpy( instr->rem, remark);
695     }
696     return instr;
697 }
698 /*****
699 iptr AllocMARK( string rem) {
700     return AllocInstr( INSTR_TOKEN_OPER, OPER_ARG_MARK, rem);
701 }
702
703 /*****
704 iptr AllocLOAD( string id, string rem) {
705     uint idx, attr;
706
707     Lookup( &did_table_desc, id, &idx, &attr);
708     return AllocInstr( INSTR_TOKEN_LOAD, idx, rem);
709 }
710 /*****
711 bptr AllocBlock( uint blen) {
712     bptr blk;
713
714     blk = ( bptr) malloc( sizeof ( block) + ( blen * sizeof ( blockentry)));
715     CheckPtr( blk);
716     blk->length = blen;
717     blk->tlength = blen; // if all members are instructions
718     return blk;
719 }
720 /*****
721 bptr CodeForall( string objref, bptr pred, bptr func) {
722     bptr blk;
723     uint objrefidx, idx, attr;
724     uint disp;
725
726     Lookup( &did_table_desc, objref, &objrefidx, &attr);
727
728     blk = AllocBlock( 13);
729     blk->bvector[ 0].bi = e_instr;
730     blk->bvector[ 0].i = AllocMARK( "forall First");
731

```

```

735     blk->bvector[ 1].bi = e_instr;
736     blk->bvector[ 1].i = AllocInstr( INSTR_TOKEN_LOAD, objrefidx, objref);
737
738     Lookup( &fnid_table_desc, "First", &idx, &attr);
739     blk->bvector[ 2].bi = e_instr;
740     blk->bvector[ 2].i = AllocInstr( INSTR_TOKEN_CALL, idx, "First");
741
742     disp = pred->tlength + func->tlength + 6;
743     blk->bvector[ 3].bi = e_instr;
744     blk->bvector[ 3].i = AllocInstr( INSTR_TOKEN_JFALSE, disp, "END.");
745
746     blk->bvector[ 4].bi = e_block;
747     blk->bvector[ 4].b = pred;
748
749     disp = func->tlength + 1;
750     blk->bvector[ 5].bi = e_instr;
751     blk->bvector[ 5].i = AllocInstr( INSTR_TOKEN_JFALSE, disp, "NEXT.");
752
753     blk->bvector[ 6].bi = e_block;
754     blk->bvector[ 6].b = func;
755
756     blk->bvector[ 7].bi = e_instr;
757     blk->bvector[ 7].i = AllocInstr( INSTR_TOKEN_OPER, OPER_ARG_POP, NULL);
758
759     blk->bvector[ 8].bi = e_instr;
760     blk->bvector[ 8].i = AllocMARK( "NEXT: forall Next");
761
762     blk->bvector[ 9].bi = e_instr;
763     blk->bvector[ 9].i = AllocInstr( INSTR_TOKEN_LOAD, objrefidx, objref);
764
765     Lookup( &fnid_table_desc, "Next", &idx, &attr);
766     blk->bvector[ 10].bi = e_instr;
767     blk->bvector[ 10].i = AllocInstr( INSTR_TOKEN_CALL, idx, "Next");
768
769     disp = pred->tlength + func->tlength + 6;
770     blk->bvector[ 11].bi = e_instr;
771     blk->bvector[ 11].i = AllocInstr( INSTR_TOKEN_JTRUE, -disp, "BODY.");
772
773     blk->bvector[ 12].bi = e_instr;
774     blk->bvector[ 12].i = AllocLOAD( "nil", "END: forall returns nil");
775
776     blk->tlength = 11 + pred->tlength + func->tlength;
777     return blk;
778 }
779 /*****
780 bptr CodeSend( string objref) {
781     bptr blk;
782     uint objrefidx, idx, attr;
783
784     blk = AllocBlock( 5);
785
786     Lookup( &did_table_desc, objref, &objrefidx, &attr);
787
788     blk->bvector[ 0].bi = e_instr;
789     blk->bvector[ 0].i = AllocMARK( "BODY: send Current");
790
791     blk->bvector[ 1].bi = e_instr;
792     blk->bvector[ 1].i = AllocInstr( INSTR_TOKEN_LOAD, objrefidx, objref);
793
794     Lookup( &fnid_table_desc, "Current", &idx, &attr);
795     blk->bvector[ 2].bi = e_instr;
796     blk->bvector[ 2].i = AllocInstr( INSTR_TOKEN_CALL, idx, "send Current");
797
798     blk->bvector[ 3].bi = e_instr;
799     blk->bvector[ 3].i = AllocInstr( INSTR_TOKEN_SEND, objrefidx, NULL);
800
801     blk->bvector[ 4].bi = e_instr;
802     blk->bvector[ 4].i = AllocMARK( "padding");
803
804     return blk;
805 }
806 /*****
807 void CodeGen( ParseTreePtr ptp) {
808

```



```

810 static char remark[ 256];
811 uint i;
812 uint idx, attr;
813 uint tlen, disp;
814 ParseTreePtr p;
815 iptr instr;
816 bptr blk;
817
818 for ( i = 0; i < ptp->NofParams; i++)
819   CodeGen( ( *ptp->Param)[ i]);
820
821 switch ( ptp->NodeAttrib) {
822   case NATTR_OBJECT_REFERENCE: {
823     ptp->info = blk = AllocBlock( 1);
824     blk->bvector[ 0].bi = e_instr;
825     blk->bvector[ 0].i = AllocLOAD( ptp->NodeLabel, ptp->NodeLabel);
826     break;
827   }
828   case NATTR_EXTERNAL_CURRENT: {
829     p = ( *ptp->Param)[ 0];
830     Lookup( &did_table_desc, p->NodeLabel, &idx, &attr);
831     strcpy( remark, p->NodeLabel);
832     if ( p->NodeAttrib == NATTR_OBJREF_FORALL_X) {
833       ptp->info = blk = AllocBlock( 1);
834       blk->bvector[ 0].bi = e_instr;
835       blk->bvector[ 0].i = AllocInstr( INSTR_TOKEN_LOAD, idx,
836                                     strcat( remark, ": Current replaced"));
837     }
838     else {
839       ptp->info = blk = AllocBlock( 3);
840       blk->bvector[ 0].bi = e_instr;
841       blk->bvector[ 0].i = AllocMARK( "Current");
842       blk->bvector[ 1].bi = e_instr;
843       blk->bvector[ 1].i = AllocInstr( INSTR_TOKEN_LOAD, idx, p->NodeLabel);
844       blk->bvector[ 2].bi = e_instr;
845       Lookup( &fnid_table_desc, ptp->NodeLabel, &idx, &attr);
846       blk->bvector[ 2].i = AllocInstr( INSTR_TOKEN_CALL, idx, "Current");
847     }
848     break;
849   }
850   case NATTR_EXTERNAL_FIRST:
851   case NATTR_EXTERNAL_NEXT:
852   case NATTR_EXTERNAL_OTHER: {
853     ptp->info = blk = AllocBlock( 2 + ptp->NofParams);
854     blk->bvector[ 0].bi = e_instr;
855     blk->bvector[ 0].i = AllocMARK( ptp->NodeLabel);
856     blk->tlength = 2;
857     for ( i = 0; i < ptp->NofParams; i++) {
858       blk->bvector[ i+1].bi = e_block;
859       p = ( *ptp->Param)[ i];
860       blk->bvector[ i+1].b = p->info;
861       blk->tlength += ( ( bptr) p->info)->tlength;
862     }
863     Lookup( &fnid_table_desc, ptp->NodeLabel, &idx, &attr);
864     blk->bvector[ ptp->NofParams+1].bi = e_instr;
865     blk->bvector[ ptp->NofParams+1].i =
866       AllocInstr( INSTR_TOKEN_CALL, idx, ptp->NodeLabel);
867     break;
868   }
869   case NATTR_INTERNAL_ASSIGN: {
870     p = ( *ptp->Param)[ 1];
871
872     ptp->info = blk = AllocBlock( 3);
873     blk->bvector[ 0].bi = e_block;
874     blk->bvector[ 0].b = p->info;
875     blk->tlength = blk->bvector[ 0].b->tlength + 2;
876
877     p = ( *ptp->Param)[ 0];
878     Lookup( &did_table_desc, p->NodeLabel, &idx, &attr);
879     blk->bvector[ 1].bi = e_instr;
880     blk->bvector[ 1].i = AllocInstr( INSTR_TOKEN_ASSIGN, idx, p->NodeLabel);
881
882     Lookup( &did_table_desc, "nil", &idx, &attr);
883     blk->bvector[ 2].bi = e_instr;
884     blk->bvector[ 2].i =

```

```

885     AllocInstr( INSTR_TOKEN_LOAD, idx, "assign returns nil");
886     break;
887   }
888   case NATTR_INTERNAL_SEQUENCE: {
889     ptp->info = blk = AllocBlock( 3 + ptp->NofParams);
890     blk->tlength = 3;
891
892     blk->bvector[ 0].bi = e_instr;
893     blk->bvector[ 0].i = AllocMARK( ptp->NodeLabel);
894
895     for ( i = 0; i < ptp->NofParams; i++) {
896       p = ( *ptp->Param)[ i];
897       blk->bvector[ i+1].bi = e_block;
898       blk->bvector[ i+1].b = p->info;
899       blk->tlength += ( ( bptr) p->info)->tlength;
900     }
901
902     blk->bvector[ ptp->NofParams+1].bi = e_instr;
903     blk->bvector[ ptp->NofParams+1].i =
904       AllocInstr( INSTR_TOKEN_OPER, OPER_ARG_CLEAR, "seq clear");
905
906     Lookup( &did_table_desc, "nil", &idx, &attr);
907     blk->bvector[ ptp->NofParams+2].bi = e_instr;
908     blk->bvector[ ptp->NofParams+2].i = AllocLOAD( "nil", "seq returns nil");
909     break;
910   }
911   case NATTR_INTERNAL_WHILE: {
912     ptp->info = blk = AllocBlock( 6);
913
914     disp = ( ( bptr) ( *ptp->Param)[ 1]->info)->tlength + 1;
915     blk->bvector[ 0].bi = e_instr;
916     blk->bvector[ 0].i = AllocInstr( INSTR_TOKEN_JUMP, disp, "while");
917
918     blk->bvector[ 1].bi = e_block;
919     blk->bvector[ 1].b = ( *ptp->Param)[ 1]->info;
920
921     blk->bvector[ 2].bi = e_instr;
922     blk->bvector[ 2].i = AllocInstr( INSTR_TOKEN_OPER, OPER_ARG_POP, NULL);
923
924     blk->bvector[ 3].bi = e_block;
925     blk->bvector[ 3].b = ( *ptp->Param)[ 0]->info;
926
927     disp += blk->bvector[ 3].b->tlength + 1;
928     blk->bvector[ 4].bi = e_instr;
929     blk->bvector[ 4].i = AllocInstr( INSTR_TOKEN_JTRUE, -disp, "AGAIN:");
930
931     blk->tlength = disp+4;
932
933     blk->bvector[ 5].bi = e_instr;
934     blk->bvector[ 5].i = AllocLOAD( "nil", "while returns nil");
935     break;
936   }
937   case NATTR_INTERNAL_IF: {
938     ptp->info = blk = AllocBlock( 7);
939
940     blk->bvector[ 0].bi = e_block;
941     blk->bvector[ 0].b = ( *ptp->Param)[ 0]->info;
942
943     blk->bvector[ 2].bi = e_block;
944     blk->bvector[ 2].b = ( *ptp->Param)[ 1]->info;
945
946     disp = blk->bvector[ 2].b->tlength + 1;
947     blk->bvector[ 1].bi = e_instr;
948     blk->bvector[ 1].i = AllocInstr( INSTR_TOKEN_JFALSE, disp, "FALSE:");
949
950     blk->bvector[ 4].bi = e_block;
951     blk->bvector[ 4].b = ( *ptp->Param)[ 2]->info;
952
953     disp = blk->bvector[ 4].b->tlength;
954     blk->bvector[ 3].bi = e_instr;
955     blk->bvector[ 3].i = AllocInstr( INSTR_TOKEN_JUMP, disp, "OUT:");
956
957     blk->bvector[ 5].bi = e_instr;
958     blk->bvector[ 5].i = AllocInstr( INSTR_TOKEN_OPER,

```



```

960         OPER_ARG_POP, "discards function result");
961
962     blk->bvector[ 6].bi = e_instr;
963     blk->bvector[ 6].i = AllocLOAD( "nil", "if returns nil");
964
965     break;
966 }
967 case NATTR_INTERNAL_FORANY: {
968     uint objrefidx;
969
970     p = ( *ptp->Param)[ 0];
971     Lookup( &did_table_desc, p->NodeLabel, &objrefidx, &attr);
972     ptp->info = blk = AllocBlock( 14);
973
974     blk->bvector[ 0].bi = e_instr;
975     blk->bvector[ 0].i = AllocMARK( "forany First");
976
977     blk->bvector[ 1].bi = e_instr;
978     blk->bvector[ 1].i = AllocInstr( INSTR_TOKEN_LOAD, objrefidx, p->NodeLabel);
979
980     Lookup( &fnid_table_desc, "First", &idx, &attr);
981     blk->bvector[ 2].bi = e_instr;
982     blk->bvector[ 2].i = AllocInstr( INSTR_TOKEN_CALL, idx, "First");
983
984     disp = 4;
985     blk->bvector[ 4].bi = e_instr;
986     blk->bvector[ 4].i = AllocInstr( INSTR_TOKEN_JUMP, disp, "TEST:");
987
988     blk->bvector[ 5].bi = e_instr;
989     blk->bvector[ 5].i = AllocMARK( "NEXT: forany Next");
990
991     blk->bvector[ 6].bi = e_instr;
992     blk->bvector[ 6].i = AllocInstr( INSTR_TOKEN_LOAD, objrefidx, p->NodeLabel);
993
994     Lookup( &fnid_table_desc, "Next", &idx, &attr);
995     blk->bvector[ 7].bi = e_instr;
996     blk->bvector[ 7].i = AllocInstr( INSTR_TOKEN_CALL, idx, "Next");
997
998     blk->bvector[ 9].bi = e_block;
999     blk->bvector[ 9].b = ( *ptp->Param)[ 1]->info;
1000
1001     blk->bvector[ 11].bi = e_block;
1002     blk->bvector[ 11].b = ( *ptp->Param)[ 2]->info;
1003
1004     blk->bvector[ 12].bi = e_instr;
1005     blk->bvector[ 12].i = AllocInstr( INSTR_TOKEN_OPER, OPER_ARG_POP,
1006                                     "discards function result");
1007
1008     blk->bvector[ 13].bi = e_instr;
1009     blk->bvector[ 13].i = AllocLOAD( "nil", "forany returns nil");
1010
1011     disp = blk->bvector[ 9].b->tlength + blk->bvector[ 11].b->tlength;
1012     blk->bvector[ 3].bi = e_instr;
1013     blk->bvector[ 3].i = AllocInstr( INSTR_TOKEN_JFALSE, disp+7, "END:");
1014
1015     blk->bvector[ 8].bi = e_instr;
1016     blk->bvector[ 8].i = AllocInstr( INSTR_TOKEN_JFALSE, disp+2, "END:");
1017
1018     disp = blk->bvector[ 9].b->tlength + 5;
1019     blk->bvector[ 10].bi = e_instr;
1020     blk->bvector[ 10].i = AllocInstr( INSTR_TOKEN_JFALSE, -disp, "NEXT:");
1021
1022     break;
1023 }
1024 case NATTR_INTERNAL_FORALL: {
1025     p = ( *ptp->Param)[ 0];
1026     ptp->info = CodeForall( p->NodeLabel,
1027                             ( *ptp->Param)[ 1]->info, ( *ptp->Param)[ 2]->info);
1028     break;
1029 }
1030 case NATTR_INTERNAL_FORALL_X: {
1031     p = ( *ptp->Param)[ 0];
1032     ptp->info = CodeForall( p->NodeLabel,
1033                             ( *ptp->Param)[ 1]->info, CodeSend( p->NodeLabel));
1034     code[ codeidx++] = ( *ptp->Param)[ 2]->info;

```

```

1035     break;
1036 }
1037 }
1038 }
1039
1040 /*****
1041 void visit( ParseTreePtr pt) {
1042     uint i;
1043
1044     printf( "%s, %d\n", pt->NodeLabel, pt->NodeAttrib);
1045     for ( i = 0; i < pt->NofParams; i++)
1046         visit( ( *pt->Param)[ i]);
1047 }
1048
1049 /*****
1050 static uint pc;
1051
1052 void CodeExtract( bptr blk) {
1053     uint i;
1054     uint tok, arg;
1055     string rem, tokenstr;
1056
1057     for ( i = 0; i < blk->blength; i++)
1058         switch ( blk->bvector[ i].bi) {
1059             case e_instr: {
1060                 printf( "(%4d): ", pc);
1061
1062                 tok = blk->bvector[ i].i->token;
1063                 arg = blk->bvector[ i].i->arg;
1064                 rem = blk->bvector[ i].i->rem;
1065
1066                 putw( tok, of);
1067                 putw( arg, of);
1068
1069                 tokenstr = tokenstrings[ tok];
1070                 if ( tok == INSTR_TOKEN_OPER) {
1071                     tokenstr = operargstrings[ arg];
1072                     printf( " %-8s", tokenstr);
1073                 }
1074                 else {
1075                     if ( jumpi( tok))
1076                         arg = pc+1+arg;
1077                     printf( " %-8s [%4d]", tokenstr, arg);
1078                 }
1079                 printf( " ;%s\n", rem == NULL ? "" : rem);
1080                 pc++;
1081                 break;
1082             }
1083             case e_block: {
1084                 CodeExtract( blk->bvector[ i].b);
1085                 break;
1086             }
1087         }
1088 }
1089
1090 /*****
1091 void oflx( void) {
1092     int i;
1093
1094     //visit( ParseTree);
1095
1096     Backbone( ParseTree);
1097     FillTables( ParseTree);
1098     JoinTables();
1099
1100     codeidx = 0;
1101     CodeGen( ParseTree);
1102     code[ codeidx] = ( bptr) ParseTree->info;
1103
1104     if ( ( of = fopen( "ofl.out", "wb")) == NULL)
1105         perror( "cannot open file");
1106
1107     putw( 'o'+f'+l', of); // marks ofl object files
1108     putw( codeidx+1, of); // number of functional units
1109     putw( ind_table_index, of); // index of first indirect object

```


A.3. EXECUTOR STRUCTURE

```

1110 putw( did_table_desc.nof_entries, of); // object table size
1111 puts( "-----");
1112 puts( "id table:");
1113 for ( i = 0; i < did_table_desc.nof_entries; i++) {
1114     printf( "%2d: %s\n", i, did_table[ i].id);
1115     fprintf( of, "%s\n", did_table[ i].id);
1116 }
1117
1118 putw( fnid_table_desc.nof_entries, of); // function table size
1119 puts( "\nfnid table:");
1120 for ( i = 0; i < fnid_table_desc.nof_entries; i++) {
1121     printf( "%2d: %s\n", i, fnid_table[ i].id);
1122     fprintf( of, "%s\n", fnid_table[ i].id);
1123 }
1124
1125 puts( "\ngenerated code:");
1126 for ( i = codeidx; i >= 0; i--) {
1127     printf( "\nfunctional unit %d, ", codeidx-i);
1128     pc = 0;
1129     putw( code[ i]->tlength, of); // length of func unit code
1130     printf( "codesize: %d\n", code[ i]->tlength);
1131     CodeExtract( code[ i]);
1132 }
1133 fclose( of);
1134 }
1135
1136 *****
1137
1138
1139

```

A.3 Executor structure

makefile

```

1140 TNUM=3
1141 TARGET=T$(TNUM)
1142 DEBUG=-g
1143 #DEBUG=
1144 PROGRESS=-i
1145
1146 ICCF=$(PROGRESS) -$(TARGET) $(DEBUG)
1147 OCCF=$(PROGRESS) -$(TARGET) $(DEBUG)
1148 ILIF=$(PROGRESS) -$(TARGET)
1149 ICCOF=$(PROGRESS) $(DEBUG)
1150 ICOLF=$(PROGRESS)
1151
1152 ICC=icc $(ICCF)
1153 OCC=cc $(OCCF)
1154 ILI=iLink $(ILIF)
1155
1156 OBJ=obj$(TARGET)/
1157
1158 all: $(OBJ)oflx.btl
1159
1160 $(OBJ)oflx.btl: $(OBJ)oflx.cfb
1161     icollect $(OBJ)oflx.cfb -o $@ $(ICOLF) #-P coll.map
1162
1163 $(OBJ)oflx.cfb: oflx.cfs $(OBJ)master.lku $(OBJ)oflx.lku\
1164     $(OBJ)dbmi.lku $(OBJ)router.lku
1165     icconf oflx.cfs -o $@ $(ICCOF)
1166
1167 $(OBJ)oflx.lku: $(OBJ)oflx.tco $(OBJ)oflx.tco
1168     $(ILI) $(OBJ)oflx.tco $(OBJ)oflx.tco -f starttrd.lnk -o $@
1169
1170 $(OBJ)oflx.tco: oflx.c oflx.h
1171     $(ICC) oflx.c -o $(OBJ)oflx.tco
1172
1173 $(OBJ)oflx.tco: oflx.c oflx.h
1174     $(ICC) oflx.c -o $(OBJ)oflx.tco
1175

```

```

1176 $(OBJ)master.lku: master.c global.h
1177     $(ICC) master.c -o $(OBJ)master.tco
1178     $(ILI) $(OBJ)master.tco -f startup.lnk -o $@
1179
1180 $(OBJ)dbmi.lku: $(OBJ)dbmi.tco $(OBJ)dbm.tco
1181     $(ILI) $(OBJ)dbmi.tco $(OBJ)dbm.tco -f starttrd.lnk -o $@
1182
1183 $(OBJ)dbmi.tco: dbmi.c dbmi.h dbm.h
1184     $(ICC) dbmi.c -o $@
1185
1186 $(OBJ)dbm.tco: dbm.c dbmi.h dbm.h
1187     $(ICC) dbm.c -o $@
1188
1189 dbmi.h: global.h
1190
1191 $(OBJ)router.lku: router.c
1192     $(ICC) router.c -o $(OBJ)router.tco
1193     $(ILI) $(OBJ)router.tco -f starttrd.lnk -o $@
1194
1195
1196

```

global.h

```

1197 #ifndef global_defined
1198 #define global_defined
1199
1200 #define DBM_BUFFER_SIZE 4096
1201
1202 #endif

```

std.h

```

1203 /* standard definitions -- 32 bit transputer version */
1204
1205 #ifndef std_h_defined
1206 #define std_h_defined
1207
1208 typedef enum {
1209     False = 0,
1210     FALSE = 0,
1211     No = 0,
1212     NO = 0,
1213     True = 1,
1214     TRUE = 1,
1215     Yes = 1,
1216     YES = 1,
1217 } bool;
1218
1219 typedef unsigned int    uint;
1220
1221 typedef unsigned char   byte;
1222
1223 typedef unsigned short  word;
1224
1225 typedef char            *string;
1226
1227 #define nop
1228
1229 #endif

```

oflx.cfs

```

1230 /*
1231  * configuration file for oflx
1232  * parallel execution environment for ofl programs
1233  */
1234
1235 val NofUnits 5;
1236
1237 /*
1238  * hardware network description
1239  */

```



```

1240 #include "boards.inc"
1241
1242 define T800( memory = 2m) PT8_2;
1243
1244 val NofPs      2*NofUnits+1;
1245
1246 PT8_2 p[ NofPs];
1247
1248 connect host,          p[ 0].link[ 3];
1249
1250 rep i=1 for NofUnits {
1251   if ( i > 1) {
1252     connect p[ i].link[ 2],      p[ i-1].link[ 1];
1253     connect p[ i+NofUnits].link[ 1],  p[ i-1+NofUnits].link[ 2];
1254   }
1255   connect p[ i].link[ 0],        p[ i+NofUnits].link[ 3];
1256   connect p[ i].link[ 3],        p[ i+NofUnits].link[ 0];
1257 }
1258 connect p[ 0].link[ 1],          p[ 1].link[ 2];
1259 connect p[ 0].link[ 2],          p[ 1+NofUnits].link[ 1];
1260
1261 /*
1262  * software network description
1263  */
1264
1265 /* server channels */
1266 input edge_FromServer;
1267 output edge_ToServer;
1268
1269 /* process types */
1270 define process(
1271   stacksize=8k, heapsize=500k,
1272   interface(
1273     int id,
1274     input inp_left,    output out_left,
1275     input inp_right,   output out_right,
1276     input inp_dbmi,    output out_dbmi,
1277     output router)
1278 ) p_oflx;
1279
1280 define process(
1281   stacksize=32k, heapsize=1000k,
1282   interface(
1283     int id,
1284     input inp_oflx,    output out_oflx,
1285     output router)
1286 ) p_dbmi;
1287
1288 define process(
1289   stacksize=32k, heapsize=500k,
1290   interface(
1291     input fs,          output ts,
1292     input inp_oflx,    output out_oflx,
1293     input router)
1294 ) p_master;
1295
1296 val RouterConnections  3;
1297
1298 define process(
1299   stacksize=32k, heapsize=500k,
1300   interface(
1301     output out,
1302     int  inpsize = RouterConnections,
1303     input inpchan[ RouterConnections])
1304 ) p_router;
1305
1306 /* processes */
1307 p_oflx      oflx[ NofUnits];
1308 p_dbmi      dbmi[ NofUnits];
1309 p_router    rout[ NofUnits];
1310 p_master    master;
1311
1312 /* process attributes */
1313 rep i=0 for NofUnits {
1314   oflx[ i]( id = NofUnits-1-i);

```

```

1315   dbmi[ i]( id = NofUnits-1-i);
1316 }
1317 rout[ NofUnits-1]( inpsize = RouterConnections-1);
1318
1319 /* process connections */
1320
1321 connect master.fs,      edge_FromServer;
1322 connect master.ts,      edge_ToServer;
1323
1324 connect master.inp_oflx,      oflx[ 0].out_left;
1325 connect master.out_oflx,      oflx[ 0].inp_left;
1326 connect master.router,        rout[ 0].out;
1327
1328 rep i=0 for NofUnits {
1329   if ( i > 0) {
1330     connect oflx[ i].inp_left,  oflx[ i-1].out_right;
1331     connect oflx[ i].out_left,  oflx[ i-1].inp_right;
1332   }
1333   connect rout[ i].out,        rout[ i-1].inpchan[ 2];
1334 }
1335 connect oflx[ i].inp_dbmi, dbmi[ i].out_oflx;
1336 connect oflx[ i].out_dbmi, dbmi[ i].inp_oflx;
1337
1338 connect dbmi[ i].router, rout[ i].inpchan[ 0];
1339 connect oflx[ i].router, rout[ i].inpchan[ 1];
1340 }
1341
1342 /* code assignment */
1343 use "objT8/oflx.lku"      for p_oflx;
1344 use "objT8/dbmi.lku"      for p_dbmi;
1345 use "objT8/master.lku"    for p_master;
1346 use "objT8/router.lku"    for p_router;
1347
1348 /*
1349  * mapping
1350  */
1351
1352 place master on p[ 0];
1353 rep i=0 for NofUnits {
1354   place oflx[ i]      on p[ i+1];
1355   place dbmi[ i]       on p[ i+1];
1356   place rout[ i]       on p[ i+1+NofUnits];
1357 }
1358
1359 place edge_FromServer on host;
1360 place edge_ToServer  on host;
1361

```

router.c

```

1362 #include "std.h"
1363 #include <stdlib.h>
1364 #include <channel.h>
1365 #include <process.h>
1366 #include <string.h>
1367 #include <misc.h>
1368
1369 static int      rqfirst, rqlast;
1370
1371 #define         RQ_SIZE 10000
1372 static string rq[ RQ_SIZE];
1373
1374 static int      nof_inputs;
1375
1376 #define         MAXINPNUM 7
1377 Channel         *inpchan[ MAXINPNUM+1];
1378 Channel         *outchan;
1379
1380 Channel         pq;      /* prod - queue */
1381 Channel         qc;      /* queue - cons */
1382
1383 string          pq_msg, qc_msg;
1384
1385 void prod( Process *p) {
1386   Channel      *c;

```



```

1387 int len;
1388
1389 p = p;
1390 while ( True) {
1391     int i;
1392
1393     i = ProcAltList( inpchan);
1394     c = inpchan[ i];
1395     pq_msg = NULL;
1396     if ( ( len = ChanInInt( c)) > 0) {
1397         ChanIn( c, pq_msg = malloc( len+1), len);
1398         pq_msg[ len] = '\0';
1399     }
1400     ChanOutInt( &pq, 0);
1401 }
1402 }
1403
1404 void buff( Process *p) {
1405     Channel *a, *b;
1406
1407     p = p;
1408     rqfirst = rqlast = 0;
1409     while ( True) {
1410
1411         if ( rqfirst == ( rqlast-1) % RQ_SIZE) /* queue full */
1412             b = NULL;
1413         else
1414             b = &pq;
1415
1416         if ( rqlast == rqfirst) { /* queue empty */
1417             a = b;
1418             b = NULL;
1419         }
1420         else
1421             a = &q;
1422
1423         switch ( ProcAlt( a, b, NULL)) {
1424             case 1:
1425                 a = b;
1426             case 0:
1427                 if ( a == &pq) {
1428                     rq[ rqfirst] = pq_msg;
1429                     rqfirst = ( rqfirst+1) % RQ_SIZE;
1430                 }
1431                 else {
1432                     qc_msg = rq[ rqlast];
1433                     rqlast = ( rqlast+1) % RQ_SIZE;
1434                 }
1435                 ChanInInt( a);
1436                 break;
1437             }
1438         }
1439     }
1440
1441 void cons( Process *p) {
1442     int len;
1443
1444     p = p;
1445     while ( True) {
1446         ChanOutInt( &q, 0);
1447         if ( qc_msg == NULL)
1448             ChanOutInt( outchan, 0);
1449         else {
1450             ChanOutInt( outchan, len = strlen( qc_msg));
1451             ChanOut( outchan, qc_msg, len);
1452             free( qc_msg);
1453         }
1454     }
1455 }
1456
1457 int main( void) {
1458     Process *p_prod, *p_buff, *p_cons;
1459     int i;
1460
1461     outchan = ( Channel *) get_param( 1);

```

```

1462
1463     nof_inputs = * ( int *) get_param( 2);
1464     if ( nof_inputs > MAXINPNUM)
1465         exit( False);
1466
1467     for ( i = 0; i < nof_inputs; i++)
1468         inpchan[ i] = ( ( Channel *) get_param( 3))[ i];
1469     inpchan[ nof_inputs] = NULL;
1470
1471     ChanInit( &pq);
1472     ChanInit( &q);
1473
1474     p_prod = ProcAlloc( prod, 0, 0);
1475     p_buff = ProcAlloc( buff, 0, 0);
1476     p_cons = ProcAlloc( cons, 0, 0);
1477
1478     ProcRun( p_prod);
1479     ProcRun( p_buff);
1480     ProcRun( p_cons);
1481
1482     ProcStop();
1483 }
1484

```

master.c

```

1485 #include <stdio.h>
1486 #include <stdlib.h>
1487 #include <string.h>
1488 #include <misc.h>
1489 #include <channel.h>
1490 #include "global.h"
1491
1492 Channel *inp_front, *out_front;
1493 Channel *inp_oflx, *out_oflx;
1494 Channel *router;
1495
1496 char dbm_buffer[ DBM_BUFFER_SIZE];
1497 int diff;
1498
1499 /*****
1500 void diff_time( void) {
1501     ChanInInt( inp_oflx);
1502     ChanInInt( inp_oflx);
1503 }
1504
1505 /*****
1506 void displays( int quits) {
1507     int nofquits;
1508     int len;
1509
1510     nofquits = 0;
1511     while ( ( quits > nofquits)) {
1512         if ( ( len = ChanInInt( router)) > 0) {
1513             ChanIn( router, dbm_buffer, len);
1514             dbm_buffer[ len] = '\0';
1515             puts( dbm_buffer);
1516         }
1517         else {
1518             nofquits++;
1519             printf( "quit: %d\n", nofquits);
1520         }
1521     }
1522 }
1523
1524 /*****
1525 int extract_dir( char *filename, char *dir) {
1526     int i, j;
1527
1528     for ( i = strlen( filename); ( i >= 0) && ( filename[ i] != '/'); --i)
1529         ;
1530
1531     for ( j = 0; j <= i; j++)
1532         dir[ j] = filename[ j];
1533

```



```

1534   dir[ i+1] = '\0';
1535   return i+1;
1536 }
1537
1538 /*****
1539 void load_dbase( char *fn) {
1540     FILE *dbf, *flist;
1541     char fname[ FILENAME_MAX];
1542     int  fnamelen, linelen, dirlen;
1543
1544     if ( ( flist = fopen( fn, "r")) == NULL) {
1545         perror( fn);
1546         exit_terminate( EXIT_FAILURE);
1547     }
1548     puts( "reading the database...");
1549     fflush( stdout);
1550
1551     dirlen = extract_dir( fn, fname);
1552
1553     while ( fgets( fname+dirlen, DBM_BUFFER_SIZE, flist) != NULL) {
1554         fnamelen = strlen( fname)-1;
1555         fname[ fnamelen] = '\0';
1556         if ( ( dbf = fopen( fname, "r")) == NULL) {
1557             perror( fname);
1558             exit_terminate( EXIT_FAILURE);
1559         }
1560
1561         puts( fname);
1562
1563         ChanOutInt( out_oflx, fnamelen = strlen( fname));
1564         ChanOut( out_oflx, dbm_buffer, fnamelen);
1565
1566         while ( fgets( dbm_buffer, DBM_BUFFER_SIZE, dbf) != NULL) {
1567             linelen = strlen( dbm_buffer)-1;
1568             dbm_buffer[ linelen] = '\0';
1569             ChanOutInt( out_oflx, linelen);
1570             ChanOut( out_oflx, dbm_buffer, linelen);
1571         }
1572         fclose( dbf);
1573         ChanOutInt( out_oflx, -1); /* negative line length => no more lines */
1574     }
1575     fclose( flist);
1576     ChanOutInt( out_oflx, 0); /*zero filename length => no more files */
1577     puts( "database read.");
1578     fflush( stdout);
1579 }
1580
1581 /*****
1582 #define getw( buffer, file)      fread( buffer, 2, 1, file)
1583
1584 void execute( char *fn) {
1585     FILE *oflf, *flist, *trace;
1586     char fname[ FILENAME_MAX];
1587     int  i, j, fnamelen, buflen, dirlen, extpos;
1588     short bufw;
1589     int  NofFuncUnits, codelen;
1590
1591     if ( ( flist = fopen( fn, "r")) == NULL) {
1592         perror( fn);
1593         exit_terminate( EXIT_FAILURE);
1594     }
1595     puts( "reading ofl programs...");
1596     fflush( stdout);
1597
1598     dirlen = extract_dir( fn, fname);
1599
1600     while ( fgets( fname+dirlen, DBM_BUFFER_SIZE, flist) != NULL) {
1601         fnamelen = strlen( fname)-1;
1602         fname[ fnamelen] = '\0';
1603         if ( ( oflf = fopen( fname, "r")) == NULL) {
1604             perror( fname);
1605             exit_terminate( EXIT_FAILURE);
1606         }
1607
1608         puts( fname);

```

```

1609     getw( &bufw, oflf); /* ofl mark */
1610     if ( bufw != 'o'+f'+l')
1611         puts( "invalid file format.");
1612     else {
1613         printf( "header, ");
1614         ChanOutInt( out_oflx, bufw);
1615         getw( &bufw, oflf); /* number of functional units */
1616         NofFuncUnits = bufw;
1617         printf( "noffuncunits: %d\n", NofFuncUnits);
1618         ChanOutInt( out_oflx, NofFuncUnits);
1619         getw( &bufw, oflf); /* index of first indirect object */
1620         ChanOutInt( out_oflx, bufw);
1621         getw( &bufw, oflf); /* object table size */
1622         ChanOutInt( out_oflx, bufw);
1623         printf( "objects (%d), ", bufw);
1624         for ( i = 0; i < bufw; i++) {
1625             fgets( dbm_buffer, DBM_BUFFER_SIZE, oflf); /* ends in "\n\0" */
1626             ChanOutInt( out_oflx, buflen = strlen( dbm_buffer)-1);
1627             ChanOut( out_oflx, dbm_buffer, buflen);
1628             dbm_buffer[ buflen] = '\0';
1629         }
1630         getw( &bufw, oflf); /* function table size */
1631         ChanOutInt( out_oflx, bufw);
1632         printf( "functions (%d), ", bufw);
1633         for ( i = 0; i < bufw; i++) {
1634             fgets( dbm_buffer, DBM_BUFFER_SIZE, oflf); /* ends in "\n\0" */
1635             ChanOutInt( out_oflx, buflen = strlen( dbm_buffer)-1);
1636             ChanOut( out_oflx, dbm_buffer, buflen);
1637         }
1638         printf( "code:");
1639         for ( j = 0; j < NofFuncUnits; j++) {
1640             getw( &bufw, oflf); /* length of func unit code */
1641             codelen = bufw;
1642             ChanOutInt( out_oflx, codelen);
1643             for ( i = 0; i < codelen; i++) {
1644                 getw( &bufw, oflf);
1645                 ChanOutInt( out_oflx, bufw);
1646                 getw( &bufw, oflf);
1647                 ChanOutInt( out_oflx, bufw);
1648             }
1649             printf( " %d", j);
1650         }
1651         puts( "\nexection output:");
1652         displays( NofFuncUnits);
1653         ChanOutInt( out_oflx, 0);
1654         ChanInInt( inp_oflx);
1655         printf( "%s executed.\n", fname);
1656
1657         extpos = strlen( fname);
1658         printf( "trace info:\n");
1659         for ( i = NofFuncUnits-1; i >= 0; i--) {
1660             int  j, eventnum;
1661             unsigned int wbuf;
1662
1663             sprintf( fname+extpos, ".trace%03d\0", i);
1664             printf( "%s: \n", fname);
1665             if ( ( trace = fopen( fname, "wb")) == NULL) {
1666                 perror( fname);
1667                 exit_terminate( EXIT_FAILURE);
1668             }
1669
1670             eventnum = ChanInInt( inp_oflx);
1671             wbuf = 32; /* time */
1672             fwrite( &wbuf, 2, 1, trace);
1673             wbuf = 8; /* tag */
1674             fwrite( &wbuf, 2, 1, trace);
1675             wbuf = 24; /* value */
1676             fwrite( &wbuf, 2, 1, trace);
1677
1678             for ( j = 0; j < eventnum; j++) {
1679                 unsigned int w;
1680
1681                 wbuf = ChanInInt( inp_oflx);
1682                 fwrite( &wbuf, 4, 1, trace);
1683                 /* printf( "time: %8d, ", wbuf); */

```


A.4. OFL ABSTRACT MACHINE

```

1684     w = ChanInInt( inp_oflx);
1685     /* printf( "tag: %2d, ", w); */
1686     wbuf = ChanInInt( inp_oflx);
1687     /* printf( "value: %8d\n", wbuf); */
1688     wbuf = ( wbuf << 8) | w;
1689     fwrite( &wbuf, 4, 1, trace);
1690 }
1691 printf( "%d\n", eventnum);
1692 fclose( trace);
1693 }
1694 }
1695 }
1696 fclose( flist);
1697 puts( "all programs executed.");
1698 fflush( stdout);
1699 }
1700
1701 /*****
1702 int main( int argc, char *argv[]) {
1703     inp_front = ( Channel*) get_param( 1);
1704     out_front = ( Channel*) get_param( 2);
1705
1706     inp_oflx = ( Channel*) get_param( 3);
1707     out_oflx = ( Channel*) get_param( 4);
1708     router = ( Channel*) get_param( 5);
1709
1710     puts( "ofldb: database manager with parallel ofl executor");
1711     fflush( stdout);
1712
1713     if ( argc != 3) {
1714         puts( "usage: ofldb dbase oflprogs");
1715         exit_terminate( EXIT_FAILURE);
1716     }
1717
1718     diff_time();
1719     load_dbase( argv[ 1]);
1720     execute( argv[ 2]);
1721
1722     exit_terminate( EXIT_SUCCESS);
1723 }
1724
1725 /*****
1726
1727

```

A.4 OFL Abstract Machine

object.h

```

1728 #ifndef object_defined
1729 #define object_defined
1730
1731 typedef struct {
1732     int indrt; /* negative values represent string lengths */
1733     char *direct;
1734 } object;
1735
1736 /* known indirect objects */
1737 #define object_false 0
1738 #define object_true 1
1739 #define object_nil 2
1740 #define object_mark 3
1741 /* values 3..1023 are reserved */
1742 #define object_reference 1024
1743
1744 #endif

```

oflx.h

```

1745 #include <channel.h>
1746 #include "global.h"

```

```

1747 #include "object.h"
1748
1749 typedef struct {
1750     int code;
1751     int arg;
1752 } instruction;
1753
1754 #include "ginstr.h"
1755
1756 extern int id;
1757 extern Channel *inp_left, *out_left;
1758 extern Channel *inp_right, *out_right;
1759 extern Channel *inp_dbmi, *out_dbmi;
1760
1761 extern char dbm_buffer[ DBM_BUFFER_SIZE];
1762
1763 extern int NofFuncUnits;
1764 extern int IndirectIndex;
1765 extern int ObjTableSize;
1766 extern int FuncTableSize;
1767
1768 extern object *ObjectTable;
1769 extern char **FunctionTable;
1770
1771 extern int CodeSize;
1772 extern instruction *CodeVector;
1773
1774 extern int diff;
1775
1776 #define MAXEVENT 1024
1777 extern int eventidx;
1778
1779 typedef struct {
1780     int time;
1781     int tag;
1782     int value;
1783 } eventrecord;
1784
1785 extern eventrecord event[ MAXEVENT];

```

oflxs.c

```

1786 #include <misc.h>
1787 #include <stdlib.h>
1788 #include <string.h>
1789 #include "std.h"
1790 #include "oflx.h"
1791
1792 void load_dbase( void) {
1793
1794     int fnamelen, linelen;
1795
1796     do {
1797         ChanOutInt( out_dbmi, fnamelen = ChanInInt( inp_left));
1798         if ( id)
1799             ChanOutInt( out_right, fnamelen);
1800         if ( fnamelen > 0) {
1801             ChanIn( inp_left, dbm_buffer, fnamelen);
1802             ChanOut( out_dbmi, dbm_buffer, fnamelen);
1803             if ( id)
1804                 ChanOut( out_right, dbm_buffer, fnamelen);
1805             do {
1806                 ChanOutInt( out_dbmi, linelen = ChanInInt( inp_left));
1807                 if ( id)
1808                     ChanOutInt( out_right, linelen);
1809                 if ( linelen >= 0) {
1810                     ChanIn( inp_left, dbm_buffer, linelen);
1811                     ChanOut( out_dbmi, dbm_buffer, linelen);
1812                     if ( id)
1813                         ChanOut( out_right, dbm_buffer, linelen);
1814                 }
1815             } while ( linelen >= 0);
1816         }
1817     } while ( fnamelen > 0);
1818 }

```



```

1819 bool oflstart( void) {
1820     int i, j, x, buflen;
1821     if ( ( x = ChanInInt( inp_left)) == 'o'+f'+l') { /* ofl mark */
1822         NofFuncUnits = ChanInInt( inp_left); /* number of func units */
1823         if ( id ) {
1824             ChanOutInt( out_right, x);
1825             ChanOutInt( out_right, NofFuncUnits > 0 ? NofFuncUnits-1 : 0);
1826         }
1827         ObjTableSize = 0;
1828         FuncTableSize = 0;
1829         if ( NofFuncUnits > 0 ) {
1830             IndirectIndex = ChanInInt( inp_left); /* index of first indirect object */
1831             ObjTableSize = ChanInInt( inp_left); /* object table size */
1832             if ( id && ( NofFuncUnits > 1) ) {
1833                 ChanOutInt( out_right, IndirectIndex);
1834                 ChanOutInt( out_right, ObjTableSize);
1835             }
1836             ObjectTable = ( object *) malloc( ObjTableSize * sizeof ( object));
1837             for ( i = 0; i < ObjTableSize; i++) {
1838                 buflen = ChanInInt( inp_left);
1839                 ChanIn( inp_left, dbm_buffer, buflen);
1840                 dbm_buffer[ buflen ] = '\0';
1841                 ObjectTable[ i ].indrcnt = -buflen;
1842                 strcpy( ObjectTable[ i ].direct = ( char *) malloc( buflen+1), dbm_buffer);
1843                 if ( id && ( NofFuncUnits > 1) ) {
1844                     ChanOutInt( out_right, buflen);
1845                     ChanOut( out_right, dbm_buffer, buflen);
1846                 }
1847             }
1848         }
1849         FuncTableSize = ChanInInt( inp_left); /* function table size */
1850         if ( id && ( NofFuncUnits > 1) ) {
1851             ChanOutInt( out_right, FuncTableSize);
1852             FunctionTable = ( char **) malloc( FuncTableSize * sizeof ( char **));
1853             for ( i = 0; i < FuncTableSize; i++) {
1854                 buflen = ChanInInt( inp_left);
1855                 ChanIn( inp_left, dbm_buffer, buflen);
1856                 dbm_buffer[ buflen ] = '\0';
1857                 strcpy( FunctionTable[ i ] = ( char *) malloc( buflen+1), dbm_buffer);
1858                 if ( id && ( NofFuncUnits > 1) ) {
1859                     ChanOutInt( out_right, buflen);
1860                     ChanOut( out_right, dbm_buffer, buflen);
1861                 }
1862             }
1863         }
1864         for ( j = NofFuncUnits; j > 0; j--) {
1865             CodeSize = ChanInInt( inp_left);
1866             if ( id && ( j > 1) ) {
1867                 ChanOutInt( out_right, CodeSize);
1868             }
1869             else {
1870                 CodeVector = ( instruction *) malloc( CodeSize * sizeof ( instruction));
1871                 for ( i = 0; i < CodeSize; i++) {
1872                     x = ChanInInt( inp_left);
1873                     if ( id && ( j > 1) ) {
1874                         ChanOutInt( out_right, x);
1875                     }
1876                     else {
1877                         CodeVector[ i ].code = x;
1878                         x = ChanInInt( inp_left);
1879                         if ( id && ( j > 1) ) {
1880                             ChanOutInt( out_right, x);
1881                         }
1882                         else {
1883                             CodeVector[ i ].arg = x;
1884                         }
1885                     }
1886                 }
1887             }
1888             return True;
1889         }
1890     }
1891     else {
1892         if ( id )
1893             ChanOutInt( out_right, x);
1894         return False;
1895     }
1896 }
1897 void oflstop( void) {

```

```

1894     int i, j;
1895     if ( NofFuncUnits > 0 ) {
1896         free( CodeVector);
1897         for ( i = 0; i < FuncTableSize; i++)
1898             free( FunctionTable[ i]);
1899         free( FunctionTable);
1900         for ( i = 0; i < ObjTableSize; i++)
1901             free( ObjectTable[ i ].direct);
1902         free( ObjectTable);
1903         for ( j = 1; j <= NofFuncUnits; j++) {
1904             ChanOutInt( out_left, eventidx);
1905             for ( i = 0; i < eventidx; i++) {
1906                 ChanOutInt( out_left, event[ i ].time);
1907                 ChanOutInt( out_left, event[ i ].tag);
1908                 ChanOutInt( out_left, event[ i ].value);
1909             }
1910             if ( j < NofFuncUnits ) {
1911                 eventidx = ChanInInt( inp_right);
1912                 for ( i = 0; i < eventidx; i++) {
1913                     event[ i ].time = ChanInInt( inp_right);
1914                     event[ i ].tag = ChanInInt( inp_right);
1915                     event[ i ].value = ChanInInt( inp_right);
1916                 }
1917             }
1918         }
1919     }
1920 }
1921 }
1922 }
1923 }
1924 }

```

oflx.c

```

1925 #include "std.h"
1926 #include <stdlib.h>
1927 #include <stdio.h>
1928 #include <string.h>
1929 #include <process.h>
1930 #include <channel.h>
1931 #include <misc.h>
1932 #include "global.h"
1933 #include "oflx.h"
1934
1935 /*****
1936 int id;
1937 Channel *inp_left, *out_left;
1938 Channel *inp_right, *out_right;
1939 Channel *inp_dbm1, *out_dbm1;
1940 Channel *router;
1941
1942 char dbm_buffer[ DBM_BUFFER_SIZE];
1943
1944 int NofFuncUnits;
1945 int IndirectIndex;
1946 int ObjTableSize;
1947 int FuncTableSize;
1948
1949 int SendSize;
1950
1951 object *ObjectTable;
1952 char **FunctionTable;
1953
1954 int CodeSize;
1955 instruction *CodeVector;
1956
1957 Channel tobuf, frombuf;
1958
1959 #define EVENT_LOAD 0
1960 #define EVENT_BUSY 1
1961
1962 int eventidx;
1963 eventrecord event[ MAXEVENT];
1964
1965 int diff;

```



```

1966 /*****
1967 void record( int tag, int value) {
1968     if ( eventidx < MAXEVENT) {
1969         event[ eventidx].time = ProcTime()-diff;
1970         event[ eventidx].tag = tag;
1971         event[ eventidx].value = value;
1972         eventidx++;
1973     }
1974 }
1975 /*****
1976 void diff_time( void) {
1977     int t;
1978     if ( ! id)
1979         diff = ProcTime();
1980     else {
1981         ChanInInt( inp_right);
1982         t = ProcTime();
1983         diff = ProcTimeMinus( t, ChanInInt( inp_right));
1984     }
1985     ChanOutInt( out_left, 0);
1986     t = ProcTime();
1987     ChanOutInt( out_left, ProcTimeMinus( t, diff));
1988 }
1989 /*****
1990 char debug_str[ 256];
1991 void debug_msg( string msg) {
1992     int l;
1993     sprintf( dbm_buffer, "%d: %s", id, msg);
1994     ChanOutInt( router, ( l = strlen( dbm_buffer)));
1995     ChanOut( router, dbm_buffer, l);
1996 }
1997 /*****
1998 #define BUFSIZE 100
1999 void oflbuffer( Process *p) {
2000     Channel *a, *b;
2001     int *buf[ BUFSIZE];
2002     int fp, lp;
2003     int i;
2004     int load;
2005     p = p;
2006     fp = lp = 0;
2007     load = 0;
2008     record( EVENT_LOAD, load);
2009     while ( True) {
2010         if ( fp == ( lp-1) % BUFSIZE) /* full */
2011             b = NULL;
2012         else
2013             b = &tobuf;
2014         if ( lp == fp) { /* empty */
2015             a = b;
2016             b = NULL;
2017         }
2018         else
2019             a = inp_left;
2020         switch( ProcAlt( a, b, NULL)) {
2021             case 1:
2022                 a = b;
2023             case 0: {
2024                 if ( a == &tobuf) {
2025                     if ( ChanInInt( a) > 0) {
2026                         record( EVENT_LOAD, ++load);
2027                     }
2028                 }
2029             }
2030         }
2031     }
2032 }
2033 /****
2034 */
2035 */
2036 */
2037 */
2038 */
2039 */
2040 */

```

```

2041     buf[ fp] = ( int *) malloc( SendSize * sizeof ( int));
2042     for ( i = 0; i < SendSize; i++)
2043         buf[ fp][ i] = ChanInInt( a);
2044     }
2045     else
2046         buf[ fp] = NULL;
2047     fp = ( fp+1) % BUFSIZE;
2048 }
2049 else {
2050     ChanInInt( inp_left);
2051     if ( buf[ lp] == NULL) {
2052         ChanOutInt( out_left, 0);
2053         ChanOutInt( &frombuf, 0); /* sync after execution */
2054     }
2055     record( EVENT_LOAD, load);
2056 }
2057 else {
2058     record( EVENT_LOAD, --load);
2059     ChanOutInt( out_left, SendSize);
2060     for ( i = 0; i < SendSize; i++)
2061         ChanOutInt( out_left, buf[ lp][ i]);
2062     free( buf[ lp]);
2063     lp = ( lp+1) % BUFSIZE;
2064 }
2065 }
2066 }
2067 }
2068 }
2069 }
2070 }
2071 }
2072 }
2073 }
2074 /*****
2075 #define STACK_SIZE 1024
2076 static object stack[ STACK_SIZE];
2077 int ip, sp;
2078 bool pop( object *obj) {
2079     if ( sp == 0)
2080         return False;
2081     else {
2082         *obj = stack[ --sp];
2083         return True;
2084     }
2085 }
2086 bool push( object obj) {
2087     if ( sp == STACK_SIZE)
2088         return False;
2089     else {
2090         stack[ sp++] = obj;
2091         return True;
2092     }
2093 }
2094 int sindex( int iobj) {
2095     int i;
2096     for ( i = sp-1; i >= 0; i--)
2097         if ( stack[ i].indrct == iobj)
2098             return i;
2099     return -1;
2100 }
2101 void am( void) {
2102     bool go;
2103     int i;
2104     instruction ci;
2105     object obj;
2106     ObjectTable[ 0].indrct = object_false;
2107     ObjectTable[ 1].indrct = object_true;
2108     ObjectTable[ 2].indrct = object_nil;
2109 }
2110 */
2111 */
2112 */
2113 */
2114 */
2115 */

```



```

2116 record( EVENT_BUSY, 0);
2117
2118 if ( NofFuncUnits > 1) {
2119     ChanOutInt( out_right, 0);
2120     go = ( ChanInInt( inp_right) > 0);
2121 }
2122 else
2123     go = Yes;
2124
2125 while ( go) {
2126     if ( NofFuncUnits > 1)
2127         for ( i = 0; i < SendSize; i++)
2128             ObjectTable[ IndirectIndex + i].indrct = ChanInInt( inp_right);
2129
2130     ip = 0;
2131     sp = 0;
2132
2133     debug_msg( "am_start");
2134
2135     record( EVENT_BUSY, 1);
2136
2137     while ( ip < CodeSize) {
2138         ci = CodeVector[ ip++];
2139
2140         switch ( ci.code) {
2141             case INSTR_TOKEN_JUMP: {
2142                 ip += ci.arg;
2143                 break;
2144             }
2145
2146             case INSTR_TOKEN_JTRUE: {
2147                 if ( pop( &obj)) {
2148                     switch ( obj.indrct) {
2149                         case object_true: {
2150                             ip += ci.arg;
2151                             break;
2152                         }
2153                         case object_false: {
2154                             nop;
2155                             break;
2156                         }
2157                         default: {
2158                             nop; /* undefined behaviour */
2159                             debug_msg( "jtrue - boolean object expected");
2160                             break;
2161                         }
2162                     }
2163                 }
2164             }
2165             else {
2166                 nop; /* undefined behaviour */
2167                 debug_msg( "jtrue - empty stack");
2168             }
2169             break;
2170         }
2171
2172         case INSTR_TOKEN_JFALSE: {
2173             if ( pop( &obj)) {
2174                 switch ( obj.indrct) {
2175                     case object_true: {
2176                         nop;
2177                         break;
2178                     }
2179                     case object_false: {
2180                         ip += ci.arg;
2181                         break;
2182                     }
2183                     default: {
2184                         nop; /* undefined behaviour */
2185                         debug_msg( "jfalse - boolean object expected");
2186                         break;
2187                     }
2188                 }
2189             }
2190             else {

```

```

2191         nop; /* undefined behaviour */
2192         debug_msg( "jtrue - empty stack");
2193     }
2194
2195     break;
2196 }
2197
2198 case INSTR_TOKEN_LOAD: {
2199     if ( ( ci.arg >= 0) && ( ci.arg < ObjTableSize)) {
2200         if ( ! push( ObjectTable[ ci.arg])) {
2201             nop; /* stack overflow */
2202             debug_msg( "load - stack overflow");
2203         }
2204     }
2205     else {
2206         nop; /* ofl translation error */
2207         debug_msg( "load - ofl translation error");
2208     }
2209
2210     break;
2211 }
2212
2213 case INSTR_TOKEN_CALL: {
2214     int markpos;
2215
2216     if ( ( ci.arg >= 0) && ( ci.arg < FuncTableSize)) {
2217         if ( ( markpos = sindex( object_mark)) < 0) {
2218             nop; /* undefined behaviour */
2219             debug_msg( "call - no mark on stack");
2220         }
2221         else {
2222             record( EVENT_BUSY, 0);
2223
2224             ChanOutInt( out_dbmi, ci.arg);
2225             ChanOutInt( out_dbmi, sp-markpos-1);
2226             while ( pop( &obj), sp > markpos)
2227                 ChanOutInt( out_dbmi, obj.indrct);
2228             obj.indrct = ChanInInt( inp_dbmi);
2229             obj.direct = NULL;
2230             push( obj);
2231
2232             record( EVENT_BUSY, 1);
2233         }
2234     }
2235
2236     }
2237
2238     break;
2239 }
2240
2241 case INSTR_TOKEN_ASSIGN: {
2242     if ( ! pop( &obj)) {
2243         nop; /* undefined behaviour */
2244         debug_msg( "assign - empty stack");
2245     }
2246     else
2247         ObjectTable[ ci.arg].indrct = obj.indrct;
2248
2249     break;
2250 }
2251
2252 case INSTR_TOKEN_SEND: {
2253     int i;
2254
2255     if ( ! pop( &obj))
2256         nop; /* undefined behaviour */
2257     else {
2258         ChanOutInt( &tobuf, 1);
2259         for ( i = 0; i < SendSize; i++)
2260             if ( i == ci.arg)
2261                 ChanOutInt( &tobuf, obj.indrct);
2262         else
2263             ChanOutInt( &tobuf, ObjectTable[ IndirectIndex+i].indrct);
2264     }
2265

```



```

2266     break;
2267 }
2268
2269 case INSTR_TOKEN_OPER: {
2270     switch (ci.arg) {
2271         case OPER_ARG_MARK: {
2272             obj.indrct = object_mark;
2273             obj.direct = NULL;
2274             if (! push( obj )) {
2275                 nop; /* stack overflow */
2276                 debug_msg( "load - stack overflow");
2277             }
2278             break;
2279         }
2280
2281         case OPER_ARG_CLEAR: {
2282             int markpos;
2283             if ( ( markpos = sindex( object_mark )) < 0 )
2284                 sp = 0;
2285             else
2286                 sp = markpos;
2287             break;
2288         }
2289
2290         case OPER_ARG_POP: {
2291             if (! pop( &obj )) {
2292                 nop; /* undefined behaviour */
2293                 debug_msg( "pop - empty stack");
2294             }
2295             break;
2296         }
2297
2298         default: {
2299             nop; /* undefined behaviour */
2300             debug_msg( "oper - invalid operation");
2301             break;
2302         }
2303     }
2304     break;
2305 }
2306
2307 default: {
2308     nop; /* undefined behaviour */
2309     debug_msg( "oflx - invalid instruction");
2310     break;
2311 }
2312 }
2313 }
2314
2315 record( EVENT_BUSY, 0);
2316
2317 if ( NofFuncUnits > 1 ) {
2318     ChanOutInt( out_right, 0);
2319     go = ( ChanInInt( inp_right ) > 0 );
2320 }
2321 else
2322     go = No;
2323 }
2324 ChanOutInt( &tobuf, 0); /* no more send */
2325 debug_msg( "am_end");
2326 }
2327
2328 /*****
2329 void execute( void ) {
2330     int i, l;
2331
2332     ChanOutInt( out_dbmi, NofFuncUnits);
2333     if ( NofFuncUnits == 0 )
2334         return;
2335
2336     ChanOutInt( out_dbmi, FuncTableSize);
2337     for ( i = 0; i < FuncTableSize; i++) {
2338         ChanOutInt( out_dbmi, l = strlen( FunctionTable[ i ] ));
2339         ChanOut( out_dbmi, FunctionTable[ i ], l);
2340     }

```

```

2341 ChanOutInt( out_dbmi, ObjTableSize);
2342 for ( i = 0; i < ObjTableSize; i++) {
2343     ChanOutInt( out_dbmi, l = -ObjectTable[ i ].indrct);
2344     ChanOut( out_dbmi, ObjectTable[ i ].direct, l);
2345 }
2346
2347 SendSize = ObjTableSize - IndirectIndex;
2348
2349 am(); /* abstract machine */
2350
2351 ChanOutInt( out_dbmi, -1); /* quit dbmi */
2352 ChanOutInt( router, 0); /* end signal to master */
2353 ChanInInt( &frombuf); /* sync */
2354 }
2355
2356 /*****
2357 bool oflstart( void);
2358 void oflstop( void);
2359 void load_dbase( void);
2360
2361 int main() {
2362     Process *p_buf;
2363
2364     id = *(int*) get_param( 1);
2365     inp_left = ( Channel*) get_param( 2);
2366     out_left = ( Channel*) get_param( 3);
2367     inp_right = ( Channel*) get_param( 4);
2368     out_right = ( Channel*) get_param( 5);
2369     inp_dbmi = ( Channel*) get_param( 6);
2370     out_dbmi = ( Channel*) get_param( 7);
2371     router = ( Channel*) get_param( 8);
2372
2373     diff_time();
2374     load_dbase();
2375
2376     ChanInit( &tobuf);
2377     ChanInit( &frombuf);
2378     p_buf = ProcAlloc( oflbuffer, 0, 0);
2379     ProcRun( p_buf);
2380
2381     while ( oflstart()) {
2382         eventidx = 0;
2383         execute();
2384         oflstop();
2385     }
2386 }
2387
2388 /****
2389
2390

```

A.5 Database Access

dbmi.h

```

2391 #ifndef dbmi_included
2392 #define dbmi_included
2393
2394 #include "global.h"
2395 #include "object.h"
2396
2397 extern char dbm_buffer[ DBM_BUFFER_SIZE];
2398 extern int id;
2399
2400 void display( void);
2401
2402 #endif

```

dbmi.c

```

2403 #include "std.h"

```



```

2404 #include <stdlib.h>
2405 #include <string.h>
2406 #include <misc.h>
2407 #include <process.h>
2408 #include <channel.h>
2409 #include "dbmi.h"
2410 #include "dbm.h"
2411
2412 int id;
2413 Channel *inp_oflx, *out_oflx;
2414 Channel *router;
2415
2416 char dbm_buffer[ DBM_BUFFER_SIZE];
2417
2418 /*****
2419 void display( void) {
2420     int messagelen;
2421
2422     if ( ( ( messagelen = strlen( dbm_buffer)) > 0) &&
2423         ( messagelen < DBM_BUFFER_SIZE-1))
2424     {
2425         ChanOutInt( router, messagelen);
2426         ChanOut( router, dbm_buffer, messagelen);
2427     }
2428 }
2429
2430 /*****
2431 void load_files( void) {
2432     int fnamelen, linelen;
2433
2434     dbm_init();
2435     do {
2436         fnamelen = ChanInInt( inp_oflx);
2437         if ( fnamelen > 0) {
2438             ChanIn( inp_oflx, dbm_buffer, fnamelen);
2439             dbm_buffer[ fnamelen] = '\0';
2440             dbm_file( fnamelen);
2441             do {
2442                 linelen = ChanInInt( inp_oflx);
2443                 if ( linelen >= 0) {
2444                     ChanIn( inp_oflx, dbm_buffer, linelen);
2445                     dbm_buffer[ linelen] = '\0';
2446                     dbm_line( linelen);
2447                 }
2448             } while ( linelen >= 0);
2449         }
2450     } while ( fnamelen > 0);
2451 }
2452
2453 /*****
2454 void execute( void) {
2455     int NofFuncNames;
2456     char **FuncNames;
2457     int NofDirObjRefs;
2458     char **DirObjRefs;
2459     int i, funcid;
2460
2461     if ( ChanInInt( inp_oflx) == 0)
2462         return;
2463
2464     NofFuncNames = ChanInInt( inp_oflx);
2465     FuncNames = ( char **) malloc( NofFuncNames * sizeof ( char *));
2466     for ( i = 0; i < NofFuncNames; i++) {
2467         int l;
2468
2469         l = ChanInInt( inp_oflx);
2470         FuncNames[ i] = ( char *) malloc( l+1);
2471         ChanIn( inp_oflx, FuncNames[ i], l);
2472         FuncNames[ i][ l] = '\0';
2473     }
2474     NofDirObjRefs = ChanInInt( inp_oflx);
2475     DirObjRefs = ( char **) malloc( NofDirObjRefs * sizeof ( char *));
2476     for ( i = 0; i < NofDirObjRefs; i++) {
2477         int l;
2478

```

```

2479         l = ChanInInt( inp_oflx);
2480         DirObjRefs[ i] = ( char *) malloc( l+1);
2481         ChanIn( inp_oflx, DirObjRefs[ i], l);
2482         DirObjRefs[ i][ l] = '\0';
2483     }
2484
2485     eval_init();
2486     while( ( funcid = ChanInInt( inp_oflx)) >= 0) {
2487         int oi, nofargs;
2488         object oo, *objs;
2489
2490         nofargs = ChanInInt( inp_oflx);
2491         objs = ( object *) malloc( nofargs * sizeof ( object));
2492         for ( i = 0; i < nofargs; i++) {
2493             oi = ChanInInt( inp_oflx);
2494             if ( oi >= 0) {
2495                 objs[ i].indirct = oi;
2496                 if ( oi < NofDirObjRefs)
2497                     objs[ i].direct = DirObjRefs[ oi];
2498             }
2499             else
2500                 objs[ i].direct = NULL;
2501         }
2502         else {
2503             objs[ i].indirct = - strlen( DirObjRefs[ -oi]);
2504             objs[ i].direct = DirObjRefs[ -oi];
2505         }
2506     }
2507     oo = eval( FuncNames[ funcid], nofargs, objs);
2508     ChanOutInt( out_oflx, oo.indrct);
2509     free( objs);
2510 }
2511
2512 for ( i = 0; i < NofFuncNames; i++)
2513     free( FuncNames[ i]);
2514 for ( i = 0; i < NofDirObjRefs; i++)
2515     free( DirObjRefs[ i]);
2516
2517 /*****
2518 int main() {
2519     id = *(int*) get_param( 1);
2520     inp_oflx = ( Channel*) get_param( 2);
2521     out_oflx = ( Channel*) get_param( 3);
2522     router = ( Channel*) get_param( 4);
2523
2524     load_files();
2525     while ( TRUE)
2526         execute();
2527 }
2528
2529 /*****
2530

```

dbm.h

```

2531 #include "dbmi.h"
2532
2533 void dbm_init( void);
2534 void dbm_file( int fnamelen);
2535 void dbm_line( int linelen);
2536
2537 void eval_init( void);
2538 object eval( const char *funcname, const int nof_args, const object args[]);
2539

```

dbm.c

```

2540 #include <process.h>
2541 #include <stdlib.h>
2542 #include <stdio.h>
2543 #include <string.h>
2544 #include "dbm.h"
2545
2546 /*****

```


A.6. TRANSLATED OFL PROGRAM

```

2547 void dbm_init( void) {
2548 }
2549
2550 /*****
2551 void dbm_file( int fnamelen) {
2552 }
2553
2554 /*****
2555 void dbm_line( int linelen) {
2556 }
2557
2558 /*****
2559 void eval_init( void) {
2560
2561     sprintf( dbm_buffer, "%d: %s", id, "eval_init");
2562     display();
2563 }
2564
2565 /*****
2566 object eval( const char *funcname, const int nof_args, const object args[]) {
2567     object obj;
2568     int i, p, pos;
2569
2570     sprintf( dbm_buffer, "%d: %s( %n" id, funcname, &pos);
2571     for ( i = 0; i < nof_args; i++) {
2572         if ( i > 0) {
2573             sprintf( dbm_buffer+pos, ", %n", &p);
2574             pos += p;
2575         }
2576         if ( args[ i].indrct < 0)
2577             sprintf( dbm_buffer+pos, "%s%n", args[ i].direct, &p);
2578         else
2579             sprintf( dbm_buffer+pos, "%d%n", args[ i].indrct, &p);
2580         pos += p;
2581     }
2582     sprintf( dbm_buffer+pos, " )");
2583     display();
2584     ProcWait( 10 + rand() % 100);
2585     obj.indrct = object_reference;
2586
2587     if ( ! strcmp( funcname, "Find"))
2588         obj.indrct = object_reference+1;
2589
2590     if ( ! strcmp( funcname, "First"))
2591         obj.indrct = object_true;
2592
2593     if ( ! strcmp( funcname, "Next"))
2594         if ( rand() % ( 6-id)*(6-id) == 0)
2595             obj.indrct = object_false;
2596         else
2597             obj.indrct = object_true;
2598
2599     if ( ! strcmp( funcname, "Current"))
2600         obj.indrct = object_reference+2;
2601
2602     if ( ! strcmp( funcname, "IntEq"))
2603         obj.indrct = object_true;
2604
2605     obj.direct = NULL;
2606
2607     return obj;
2608 }
2609
2610 }
2611
2612 /*****

```

```

2614 id table:
2615 0: false
2616 1: true
2617 2: nil
2618 3: People
2619 4: age
2620 5: 16
2621 6: owner
2622 7: composed
2623 8: lastname
2624 9: color
2625 10: label
2626 11: _person
2627 12: _vehicle
2628 13: _part
2629
2630 fnid table:
2631 0: Current
2632 1: First
2633 2: Next
2634 3: IntEq
2635 4: Field
2636 5: Display
2637
2638 generated code:
2639
2640 functional unit #0, codesize: 29
2641 ( 0): mark ;sequence
2642 ( 1): load [ 3] ;People
2643 ( 2): assign [ 11] ;_person
2644 ( 3): load [ 2] ;assign returns nil
2645 ( 4): mark ;forall First
2646 ( 5): load [ 11] ;_person
2647 ( 6): call [ 1] ;First
2648 ( 7): jfalse [ 26] ;END:
2649 ( 8): mark ;IntEq
2650 ( 9): mark ;Field
2651 ( 10): load [ 4] ;age
2652 ( 11): load [ 11] ;_person: Current replaced
2653 ( 12): call [ 4] ;Field
2654 ( 13): load [ 5] ;16
2655 ( 14): call [ 3] ;IntEq
2656 ( 15): jfalse [ 22] ;NEXT:
2657 ( 16): mark ;BODY: send Current
2658 ( 17): load [ 11] ;_person
2659 ( 18): call [ 0] ;send Current
2660 ( 19): send [ 11] ;;
2661 ( 20): mark ;padding
2662 ( 21): pop ;;
2663 ( 22): mark ;NEXT: forall Next
2664 ( 23): load [ 11] ;_person
2665 ( 24): call [ 2] ;Next
2666 ( 25): jtrue [ 8] ;BODY:
2667 ( 26): load [ 2] ;END: forall returns nil
2668 ( 27): clear ;seq clear
2669 ( 28): load [ 2] ;seq returns nil
2670
2671 functional unit #1, codesize: 26
2672 ( 0): mark ;sequence
2673 ( 1): mark ;Field
2674 ( 2): load [ 6] ;owner
2675 ( 3): load [ 11] ;_person: Current replaced
2676 ( 4): call [ 4] ;Field
2677 ( 5): assign [ 12] ;_vehicle
2678 ( 6): load [ 2] ;assign returns nil
2679 ( 7): mark ;forall First
2680 ( 8): load [ 12] ;_vehicle
2681 ( 9): call [ 1] ;First
2682 ( 10): jfalse [ 23] ;END:
2683 ( 11): load [ 1] ;true
2684 ( 12): jfalse [ 19] ;NEXT:
2685 ( 13): mark ;BODY: send Current
2686 ( 14): load [ 12] ;_vehicle
2687 ( 15): call [ 0] ;send Current
2688 ( 16): send [ 12] ;;

```

A.6 Translated OFL program

```

demo.out
2613 -----

```



```

2689 ( 17): mark           ;padding
2690 ( 18): pop           ;
2691 ( 19): mark           ;NEXT: forall Next
2692 ( 20): load           [ 12] ;_vehicle
2693 ( 21): call           [ 2] ;Next
2694 ( 22): jtrue          [ 11] ;BODY:
2695 ( 23): load           [ 2] ;END: forall returns nil
2696 ( 24): clear          ;seq clear
2697 ( 25): load           [ 2] ;seq returns nil
2698
2699 functional unit #2, codesize: 26
2700 ( 0): mark             ;sequence
2701 ( 1): mark             ;Field
2702 ( 2): load             [ 7] ;composed
2703 ( 3): load             [ 12] ;_vehicle: Current replaced
2704 ( 4): call             [ 4] ;Field
2705 ( 5): assign           [ 13] ;_part
2706 ( 6): load             [ 2] ;assign returns nil
2707 ( 7): mark             ;forall First
2708 ( 8): load             [ 13] ;_part
2709 ( 9): call             [ 1] ;First
2710 (10): jfalse           [ 23] ;END:
2711 (11): load             [ 1] ;true
2712 (12): jfalse           [ 19] ;NEXT:
2713 (13): mark             ;BODY: send Current
2714 (14): load             [ 13] ;_part
2715 (15): call             [ 0] ;send Current
2716 (16): send            [ 13] ;
2717 (17): mark             ;padding
2718 (18): pop             ;
2719 (19): mark             ;NEXT: forall Next
2720 (20): load             [ 13] ;_part
2721 (21): call             [ 2] ;Next
2722 (22): jtrue            [ 11] ;BODY:
2723 (23): load             [ 2] ;END: forall returns nil
2724 (24): clear            ;seq clear
2725 (25): load             [ 2] ;seq returns nil
2726
2727 functional unit #3, codesize: 21
2728 ( 0): mark             ;sequence
2729 ( 1): mark             ;Display
2730 ( 2): mark             ;Field
2731 ( 3): load             [ 8] ;lastname
2732 ( 4): load             [ 11] ;_person: Current replaced
2733 ( 5): call             [ 4] ;Field
2734 ( 6): call             [ 5] ;Display
2735 ( 7): mark             ;Display
2736 ( 8): mark             ;Field
2737 ( 9): load             [ 9] ;color
2738 (10): load             [ 12] ;_vehicle: Current replaced
2739 (11): call             [ 4] ;Field
2740 (12): call             [ 5] ;Display
2741 (13): mark             ;Display
2742 (14): mark             ;Field
2743 (15): load             [ 10] ;label
2744 (16): load             [ 13] ;_part: Current replaced
2745 (17): call             [ 4] ;Field
2746 (18): call             [ 5] ;Display
2747 (19): clear            ;seq clear
2748 (20): load             [ 2] ;seq returns nil

```


Appendix B

Load balancing test system code listing

```
makefile
1 #####
2 # processor commune makefile
3
4 # unix: make X=- Y=/
5 # dos: make X=/ Y=\
6 #
7 #####
8 X=-
9 Y=/
10 TNUM      =8
11 TARGET    =T$(TNUM)
12 #DEBUG    =$(Xy)
13 DEBUG     =
14 MODE      =h
15 #OCCF     =$(Xi) $(X$(TARGET)) $(DEBUG) $(Xx)
16 OCCF      =$(Xi) $(X$(TARGET)) $(DEBUG) $(XWO)
17 #ILINKF   =$(Xi) $(X$(TARGET)) $(DEBUG) $(X$(MODE))
18 ILINKF    =$(Xi) $(X$(TARGET)) $(DEBUG)
19 OCCONFF   =$(Xi) $(DEBUG) $(X$(MODE))
20 ICOLLF    =$(Xi)
21 O         =obj$Y
22 C         =config$Y
23 P         =profile$Y
24 B         =boss$Y
25 A         =app$Y
26 S         =store$Y
27 OCC       =oc $(OCCF)
28 ILINK     =ilink $(ILINKF)
29 CONF      =occonf $(OCCONFF)
30
31 all:      $Ocommune.btl
32           @sepline
33
34 #####
35 # CONFIG & COLLECT
36 #####
37 $Ocommune.btl:  $Ccommune.pgm $Ocmnroot.lku $Ocommune.lku
38               @sepline
39               $(CONF) $Ccommune.pgm $Xo $Ocommune.cfb
40               @sepline
41               icollect $(ICOLLF) $Ocommune.cfb $Xo $Xp commune.map
42
43 #####
44 # LINK
45 #####
```

```
46 $Ocmnroot.lku:  $Ocmnroot.tco
47               @sepline
48               $(ILINK) $Ocmnroot.tco $Ocmnnpars.tco hostio.lib convert.lib \
49               string.lib $Xf occam$(TNUM).lnk $Xo $O
50
51 $Ocommune.lku:  $Ocommune.tco $Oprofile.tco $Oapp.tco
52               @sepline
53               $(ILINK) $Xf commune.lnk snglmath.lib $Xf occam$(TNUM).lnk $Xo $O
54
55 #####
56 # OCC
57 #####
58 $Ocmnroot.tco:  $Ccommune.inc cmnroot.occ $Ocmnnpars.tco
59               @sepline
60               $(OCC) cmnroot.occ $Xo $O
61
62 $Ocmnnpars.tco:  cmnnpars.occ
63               @sepline
64               $(OCC) cmnnpars.occ $Xo $O
65
66 $Ocommune.tco:  $Ccommune.inc commune.occ $Ocmnmain.tco \
67               $Oprofile.tco $Pprofserv.inc $Oprofserv.tco
68               @sepline
69               $(OCC) commune.occ $Xo $O
70
71 $Ocmnmain.tco:  $Ccommune.inc cmnmain.occ $Osentinel.tco \
72               $Bcmnboss.inc $Ocmnboss.tco \
73               $Scmnstor.inc $Ocmnstor.tco \
74               $Pprofserv.inc
75               @sepline
76               $(OCC) cmnmain.occ $Xo $O
77
78 $Osentinel.tco:  sentinel.occ $Ccommune.inc $Bcmnboss.inc \
79               $Scmnstor.inc $Oguard.tco
80               @sepline
81               $(OCC) sentinel.occ $Xo $O
82
83 $Oguard.tco:    guard.occ $Bcmnboss.inc $Scmnstor.inc $Ostorifac.tco
84               @sepline
85               $(OCC) guard.occ $Xo $O
86
87
88 # PROFILE #####
89 $Oprofile.tco:  $Ccommune.inc $Pprofile.occ $Oprofdiff.tco
90               @sepline
91               $(OCC) $Pprofile.occ $Xo $O
92
```



```

93 $0profdiff.tco: $Pprofdiff.occ
94     @sepline
95     $(OCC) $Pprofdiff.occ $Xo $0
96
97 $0profserv.tco: $Pprofserv.occ
98     @sepline
99     $(OCC) $Pprofserv.occ $Xo $0
100
101 # STORE #####
102 $0cmmnstor.tco: $Scmmnstor.occ $Scmmnstor.inc $Ccommune.inc $Pprofserv.inc
103     @sepline
104     $(OCC) $Scmmnstor.occ $Xo $0
105
106 $0storifac.tco: $Sstorifac.occ $Scmmnstor.inc
107     @sepline
108     $(OCC) $Sstorifac.occ $Xo $0
109
110 # BOSS #####
111 $0cmmnboss.tco: $Bcmmnboss.occ $Bcmmnboss.inc \
112     $Scmmnstor.inc $Pprofserv.inc \
113     $0feeder.tco \
114     $Aappifac.inc $0app.tco
115     @sepline
116     $(OCC) $Bcmmnboss.occ $Xo $0
117
118 $0feeder.tco: $Bfeeder.occ $Bcmmnboss.inc \
119     $Scmmnstor.inc $Pprofserv.inc \
120     $Aappifac.inc $0app.tco
121     @sepline
122     $(OCC) $Bfeeder.occ $Xo $0
123
124 # APP #####
125 $0app.tco: $Aapp.occ $Aappifac.inc
126     @sepline
127     $(OCC) $Aapp.occ $Xo $0

```

commune.lnk

```

128 obj/commune.tco
129 obj/cmmnmain.tco
130 obj/profile.tco
131 obj/profdiff.tco
132 obj/profserv.tco
133 obj/sentinel.tco
134 obj/guard.tco
135 obj/cmmnstor.tco
136 obj/storifac.tco
137 obj/cmmnboss.tco
138 obj/feeder.tco
139 obj/app.tco
140 debug.lib
141 convert.lib

```

B.1 Configuration

config/commune.inc

```

142 -----
143 --- commune.inc --- global declarations for processor commune
144 -----
145
146 --- refer to inivec, outvec in commune.pgm
147 VAL HamBwSide ..... IS ... 1: --- channel index Hamilton backward
148 VAL HamFwSide ..... IS ... 0: --- channel index Hamilton forward
149
150 VAL MaxNameLen ..... IS ... 256: --- file names
151
152 VAL MaxNofNeighbours IS ... 4: --- nof transputer links
153
154 -----

```

APPENDIX B. LOAD BALANCING TEST SYSTEM CODE LISTING

config/commune.pgm

```

155 -----
156 --- Commune configuration file (ring + cross connections)
157 -----
158
159 VAL KILO IS 1024:
160 VAL MEGA IS KILO*KILO:
161
162 -----
163 --- hardware specific values
164 --- Telmat.T.Node.16 transputers T8-2M
165 -----
166
167 #INCLUDE "osntab.inc"
168
169 VAL TPUTER.TYPE ..... IS "T800":
170 VAL MEMSIZE.ROOT ..... IS SN.HST[ 0][ SN.memory]:
171
172 VAL LIDX.HOSTLINK ..... IS 0: ..... root to host
173 VAL LIDX.ROOTLINK ..... IS SN.EDGE.HOST: ..... root to head
174 VAL LIDX.HEADLINK ..... IS SN.EDGE.TNODE: ..... head to root
175 VAL LIDX.FORWARD ..... IS 1: ..... not for tail
176 VAL LIDX.CLOSETAIL ..... IS 1: ..... forward for tail
177 VAL LIDX.BACKWARD ..... IS 2: ..... not for head
178 VAL LIDX.CLOSEHEAD ..... IS 2: ..... backward for head
179 VAL LIDX.CROSS.1 ..... IS 0: ..... crosslink one side
180 VAL LIDX.CROSS.2 ..... IS 3: ..... crosslink other side
181
182 VAL NofNodes ..... IS 16: --- on top of root
183
184 -----
185 --- physical processors
186 -----
187
188
189 NODE ..... root:
190 [ NofNodes] NODE node:
191
192 head IS node[ 0]:
193 tail IS node[ NofNodes-1]:
194
195 ARC ..... hostlink:
196 ARC ..... rootlink:
197 [ NofNodes] ARC ringlink:
198 [ NofNodes/2] ARC crosslink:
199
200 NETWORK
201 .. DO
202 ..... node attributes
203 ..... SET root( type, memsize := TPUTER.TYPE, MEMSIZE.ROOT)
204 ..... DO i=0 FOR NofNodes
205 ..... SET node[ i]( type, memsize := TPUTER.TYPE, SN.WRK[ i][ SN.memory))
206
207 --- host connection
208 ..... CONNECT root[ link][ LIDX.HOSTLINK].TO
209 ..... HOST ..... WITH hostlink
210 ..... CONNECT root[ link][ LIDX.ROOTLINK].TO
211 ..... head[ link][ LIDX.HEADLINK].WITH rootlink
212
213 --- ring
214 ..... DO i=0 FOR NofNodes-1
215 ..... CONNECT node[ i][ link][ LIDX.FORWARD].TO
216 ..... node[ i+1][ link][ LIDX.BACKWARD].WITH ringlink[ i]
217
218 ..... CONNECT head[ link][ LIDX.CLOSEHEAD].TO
219 ..... tail[ link][ LIDX.CLOSETAIL].WITH ringlink[ NofNodes-1]
220
221 --- cross connections
222 ..... DO i=0 FOR NofNodes/2
223 ..... CONNECT node[ i][ link][ LIDX.CROSS.1].TO
224 ..... node[ i+( NofNodes/2)][ link][ LIDX.CROSS.2].WITH crosslink[ i]
225 :
226
227 #INCLUDE "hostio.inc"

```


B.2. HOST CONNECTION

```

228 #USE....."obj/cmmnroot.lku"
229 #USE....."obj/commune.lku"
230
231 -----
232 --- logical processors
233 -----
234
235 NODE.....Root:
236 [. NofNodes]NODE...Node:
237
238 CONFIG
239 ..CHAN OF SP...fs, ts:
240 ..PLACE.....fs, ts ON hostlink:
241
242 ..[. NofNodes]CHAN OF INT...HamFw,...HamBw:--- Hamiltonian cycle on ring
243
244 ..[. NofNodes]CHAN OF INT...Cross:--- channels for cross connect
245
246 ..[. NofNodes]CHAN OF INT...EnvInp,...EnvOut:--- channels to outside world
247
248 ..PLACE EnvOut[. 0], EnvInp[. 0] ON rootlink:
249
250 ..PAR
251 .....PROCESSOR Root
252 .....cmmnroot(. fs, ts, EnvOut[. 0], EnvInp[. 0], NofNodes)
253
254 .....PAR i=0 FOR NofNodes
255 .....PLACE HamFw[i], HamBw[i] ON ringlink[i]:
256 .....PLACE Cross[i].....ON crosslink[i REM (NofNodes/2)]:
257
258 .....PROCESSOR Node[i]
259 .....fw.inp...IS HamFw[(. (i+NofNodes)-1) REM NofNodes]:
260 .....fw.out...IS HamFw[i]:
261
262 .....bw.inp...IS HamBw[i]:
263 .....bw.out...IS HamBw[(. (i+NofNodes)-1) REM NofNodes]:
264
265 .....env.inp...IS EnvInp[i]:
266 .....env.out...IS EnvOut[i]:
267
268 .....[MEGA/3]INT...freememory:
269
270 .....-- ref to inpvec, outvec indeces in commune.inc
271 .....-- HamFwSide, HamBwSide
272 .....inpvec IS [. bw.inp, fw.inp, Cross[i]]:
273 .....outvec IS [. fw.out, bw.out, Cross[(i+(NofNodes/2)) REM NofNodes]]:
274 .....commune(i, inpvec, outvec, env.inp, env.out, freememory)
275 :
276
277 -----
278 --- mapping
279 -----
280
281 VAL high IS 0:
282 VAL low IS 1:
283 MAPPING
284 ..DO
285 ...MAP Root ONTO root PRI low
286 ...DO i=0 FOR NofNodes
287 ...MAP Node[i] ONTO node[i] PRI low
288 :

```

B.2 Host connection

```
cmmnroot.occ
```

```

289 #INCLUDE....."hostio.inc"
290 #USE....."hostio.lib"
291 #USE....."debug.lib"
292 #USE....."convert.lib"
293 #INCLUDE....."config/commune.inc"

```

```

294
295 PROC cmmnroot(. CHAN OF SP...fs, ts,
296 .....CHAN OF INT...NetworkInp, NetworkOut,
297 .....VAL INT...NofNodes)
298
299
300 --- sizes in INT units
301 ..INT.....ProfileSize:--- profile buffer size, 0=no profiling
302 ..INT.....EvalSize:--- evaluator buffer size
303 ..INT.....InitDataSize:--- evaluator init data size
304 ..INT.....MaxTaskSize:
305 ..INT.....RootTaskSize:
306
307 ..INT32.....fid.data, fid.task:
308 [. MaxNameLen]BYTE...OutDirName:
309 ..INT.....OutDirLen:
310
311 ..PROC putsnl(. VAL [BYTE] string)
312 .....so.write.string.nl(. fs, ts, string)
313 :
314 ..PROC puts(. VAL [BYTE] string)
315 .....so.write.string(. fs, ts, string)
316 :
317 ..PROC puti(. VAL INT n)
318 .....so.write.int(. fs, ts, n, 0)
319 :
320 ..PROC putx(. VAL INT n)
321 .....so.write.hex.int(. fs, ts, n, 8)
322 :
323 ..PROC putsnl(. VAL [BYTE] string, VAL INT n)
324 .....SEQ
325 .....puts(. string)
326 .....puti(. n)
327 .....putsnl(. "")
328 :
329 ..PROC writeint(. VAL INT32 fid, VAL INT n) --
330 .....INT x, len:
331 .....SEQ
332 .....x := n
333 .....[BYTE] int RETYPES x:
334 .....so.write(. fs, ts, fid, int, len)
335 .....writeint:-----
336
337 ..PROC profSetup()-----
338 .....INT nodecnt:
339 .....SEQ
340 .....NetworkOut ! ProfileSize
341 .....puts(. "profile flag...")
342 .....NetworkInp ? nodecnt
343 .....putsnl(. "distributed.")
344 .....puts(. "hamilton cycle check:")
345 .....IF
346 .....nodecnt = NofNodes
347 .....SEQ
348 .....puts(. "passed. (")
349 .....puti(. NofNodes)
350 .....putsnl(. "nodes)")
351 .....TRUE
352 .....SEQ
353 .....puti(. nodecnt)
354 .....putsnl(. "failed.")
355 .....so.exit(. fs, ts, sps.failure)
356 .....DEBUG ASSERT(. FALSE)
357 :--- profSetup-----
358
359 ..PROC profClose()-----
360 .....SEQ
361 .....puts(. "profile data flush...")
362
363 .....VAL ProfExt IS ".cpf":
364 .....VAL ProfLen IS 8:
365 .....VAL xlen...IS OutDirLen+ProfLen:
366 .....[MaxNameLen]BYTE profname:
367 .....[ProfLen]BYTE...ProfFileName:
368 .....INT...numlen:

```



```

369 ..... INT32..pfid:
370 ..... BYTE..result:
371 ..... SEQ
372 ..... [ profname FROM 0 FOR OutDirLen] := [ OutDirName FROM 0 FOR OutDirLen]
373 ..... SEQ pnum=0 FOR NofNodes
374 ..... INT n, cnt:
375 ..... SEQ
376 ..... HEXTOSTRING(. numlen, . ProfFileName, . pnum)
377 ..... [ profname FROM OutDirLen FOR 8] := "00000000"
378 ..... [ profname FROM xlen- numlen FOR numlen] := [ ProfFileName FROM 0 FOR numlen]
379 ..... [ profname FROM xlen- numlen FOR SIZE ProfExt] := ProfExt
380 ..... so.open(. fs, . ts, [ profname FROM 0 FOR xlen+ (SIZE ProfExt)],
381 ..... spt.binary, . spm.output, . pfid, . result)
382 .....
383 ..... NetworkInp.? n --- skip
384 .....
385 ..... SEQ i=0 FOR 5
386 ..... SEQ
387 ..... NetworkInp.? n
388 ..... writeint(. pfid, n)
389 ..... NetworkInp.? cnt
390 ..... SEQ i=0 FOR cnt
391 ..... SEQ
392 ..... NetworkInp.? n
393 ..... writeint(. pfid, n)
394 .....
395 ..... so.close(. fs, . ts, . pfid, . result)
396 ..... puti(. pnum)
397 .....
398 ..... putsnl(. "...ok.")
399 ..... --- profClose:-----
400 .....
401 ..... PROC phase2()-----
402 ..... SEQ
403 ..... NetworkOut.! MaxTaskSize
404 ..... NetworkOut.! RootTaskSize
405 ..... NetworkOut.! EvalSize
406 ..... NetworkOut.! InitDataSize
407 .....
408 ..... CHAN OF BYTE NetworkOut RETYPES NetworkOut:
409 ..... CHAN OF BYTE NetworkInp RETYPES NetworkInp:
410 .....
411 ..... PROC load(. VAL INT32.fid)-----
412 ..... INT length:
413 ..... [ 4096] BYTE buf:
414 ..... BOOL loading:
415 ..... SEQ
416 ..... loading := TRUE
417 ..... WHILE loading
418 ..... SEQ
419 ..... so.read(. fs, . ts, . fid, . length, . buf)
420 ..... SEQ i=0 FOR length
421 ..... NetworkOut.! buf[i]
422 ..... loading := length = (. SIZE buf)
423 ..... --- load:-----
424 ..... BYTE result:
425 ..... SEQ
426 ..... --- root task load
427 ..... puts(. "loading task...")
428 ..... load(. fid.task)
429 ..... so.close(. fs, . ts, . fid.task, . result)
430 ..... putsnl(. "ok.")
431 .....
432 ..... --- data distribution
433 ..... puts(. "data distribution...")
434 ..... load(. fid.data)
435 ..... so.close(. fs, . ts, . fid.data, . result)
436 ..... putsnl(. "ok.")
437 .....
438 ..... --- processing
439 ..... puts(. "*nprocessing...")
440 ..... INT dummy:
441 ..... NetworkInp.? dummy
442 ..... putsnl(. "terminated*n")
443 ..... --- end processing

```

APPENDIX B. LOAD BALANCING TEST SYSTEM CODE LISTING

```

444 .....
445 ..... --- result.flush
446 ..... puts(. "flushing results...")
447 ..... VAL ext.... IS ".cpr":
448 ..... VAL saveLen IS 8:
449 ..... VAL xlen.... IS OutDirLen+saveLen:
450 ..... [ MaxNameLen] BYTE saveName:
451 ..... [ saveLen] BYTE saveFileName:
452 .....
453 ..... INT.... numlen:
454 ..... INT32.. fid:
455 ..... BYTE.. result:
456 ..... SEQ
457 ..... [ saveName FROM 0 FOR OutDirLen] := [ OutDirName FROM 0 FOR OutDirLen]
458 ..... SEQ pnum=0 FOR NofNodes
459 ..... INT n, cnt:
460 ..... SEQ
461 ..... HEXTOSTRING(. numlen, . saveFileName, . pnum)
462 ..... [ saveName FROM OutDirLen FOR 8] := "00000000"
463 ..... [ saveName FROM xlen- numlen FOR numlen] := [ saveFileName FROM 0 FOR numlen]
464 ..... [ saveName FROM xlen- numlen FOR SIZE ext] := ext
465 ..... so.open(. fs, . ts, [ saveName FROM 0 FOR xlen+ (SIZE ext)],
466 ..... spt.binary, . spm.output, . fid, . result)
467 .....
468 ..... NetworkInp.? n
469 .....
470 ..... CHAN OF BYTE NetworkInp RETYPES NetworkInp:
471 ..... [ 4096] BYTE buf:
472 ..... INT cnt, bcnt:
473 ..... SEQ
474 ..... cnt := 0
475 ..... WHILE cnt < n
476 ..... SEQ
477 ..... bcnt := 0
478 ..... WHILE (. bcnt < (. SIZE buf)) AND (. cnt < n)
479 ..... SEQ
480 ..... NetworkInp.? buf[. bcnt]
481 ..... cnt, bcnt := cnt+1, bcnt+1
482 .....
483 ..... INT written:
484 ..... so.write(. fs, . ts, . fid, [ buf FROM 0 FOR bcnt], . written)
485 .....
486 ..... puti(. pnum)
487 ..... BYTE result:
488 ..... so.close(. fs, . ts, . fid, . result)
489 .....
490 ..... INT dummy:
491 ..... NetworkInp.? dummy
492 ..... putsnl(. "...ok.")
493 ..... --- end result.flush
494 .....
495 ..... --- phase2:-----
496 ..... SEQ
497 ..... putsnl(. "--- Processor Commune ---*n")
498 ..... #USE "obj/cmmnpar.tco"
499 ..... cmmnpar(. fs, . ts,
500 ..... ProfileSize,
501 ..... EvalSize,
502 ..... MaxTaskSize,
503 ..... InitDataSize,
504 ..... RootTaskSize,
505 ..... fid.data, . fid.task,
506 ..... OutDirName, . OutDirLen)
507 .....
508 ..... IF
509 ..... ProfileSize > 0
510 ..... putsnl(. "profiling enabled,*n buffer size=*t", . ProfileSize)
511 ..... TRUE
512 ..... putsnl(. "profiling disabled")
513 .....
514 ..... putsnl(. "maxtasksz=*t", . MaxTaskSize)
515 ..... putsnl(. "roottasksz=*t", . RootTaskSize)
516 ..... putsnl(. "evalsize...=*t", . EvalSize)
517 ..... putsnl(. "initdatasz=*t", . InitDataSize)
518 .....

```



```

519... putsnl(·"nsetup sequence started.")
520
521... profSetup()--- profile flag distribution, hamilton test
522
523... phase2()
524
525... IF
526... ProfileSize > 0
527... profClose()--- profile data flush
528... TRUE
529... SKIP
530
531... INT x:
532... NetworkInp ? x
533... putsnl(·"ready.")
534... so.exit(·fs, ts, sps.success)
535:--- cmnnroot-----

```

cmnnpars.occ

```

536 #INCLUDE..... "hostio.inc"
537 #USE..... "hostio.lib"
538 #USE..... "string.lib"
539 #INCLUDE..... "config/commune.inc"
540
541 VAL dirSeparator IS '\/':
542
543 PROC cmnnpars(·CHAN OF SP... fs, ts,
544... INT... ProfileSize,
545... INT... EvalSize,
546... INT... MaxTaskSize,
547... INT... InitDataSize,
548... INT... RootTaskSize,
549... INT32... fid.data, fid.task,
550... [BYTE... OutDirName,
551... INT... OutDNameLen)
552
553... PROC condtest(·VAL BOOL condition, VAL [BYTE msg1, msg2)-----
554... IF
555... condition
556... SEQ
557... so.write.string(·fs, ts, msg1)
558... so.write.string.nl(·fs, ts, msg2)
559... so.exit(·fs, ts, sps.failure)
560... STOP
561... TRUE
562... SKIP
563:--- end: condtest-----
564
565--- Global variables-----
566
567... [·MaxNameLen]BYTE... DescFileName,..... name of descriptor file
568... DataFileName,..... name of data file
569... TaskFileName,..... name of task file
570... OutFileName:..... name of out file (·results, ·log data)
571
572... INT... DescFNameLen, DataFNameLen:
573... INT... TaskFNameLen, OutFNameLen:
574
575... BYTE... result:
576... INT32... DescFileId:
577... BOOL... success:
578
579
580... PROC ParseCmdLine()-----
581... PROC ExtractName(·VAL [BYTE line, --- path string to parse
582... INT start, --- start position of name
583... INT len, --- length of name
584... INT lastper) --- position of last dirSeparator
585... BOOL found:
586... SEQ
587... WHILE (·start < (·SIZE line)) AND (·line[·start] = 's')
588... start := start+1
589... len, lastper, found := 0, 0, FALSE
590... WHILE (·start+len) < (·SIZE line) AND (·NOT found)

```

```

591... CASE line[·start+len]
592... dirSeparator
593... SEQ
594... lastper := len
595... len := len+1
596... 's'
597... found := TRUE
598... ELSE
599... len := len+1
600:--- end: ExtractName-----
601
602... INT... len, fnamestart, lastper, dirstart, dirper:
603... [·MaxNameLen]BYTE... line:
604... SEQ
605... so.commandline(·fs, ts, sp.short.commandline, len, line, result)
606... condtest(·result <> spr.ok, "invalid command line", "")
607
608... cmdline IS [·line FROM 0 FOR len]:
609... SEQ
610... fnamestart := 0
611... ExtractName(·cmdline, fnamestart, DescFNameLen, lastper)
612... condtest(·DescFNameLen = 0, "missing input file name", "")
613... [·DescFileName FROM 0 FOR DescFNameLen] :=
614... [·cmdline FROM fnamestart FOR DescFNameLen]
615
616... dirstart := fnamestart+DescFNameLen
617... ExtractName(·cmdline, dirstart, OutDNameLen, dirper)
618... IF
619... OutDNameLen = 0
620... SEQ
621... dirstart := fnamestart
622... OutDNameLen := lastper
623... TRUE
624... SKIP
625... [·OutDirName FROM 0 FOR OutDNameLen] :=
626... [·cmdline FROM dirstart FOR OutDNameLen]
627
628... IF
629... OutDNameLen = 0
630... SEQ
631... OutDNameLen := 1
632... OutDirName[0] := ' '
633... TRUE
634... SKIP
635... OutDirName[·OutDNameLen] := dirSeparator
636... OutDNameLen := OutDNameLen + 1
637
638... so.write.string(·fs, ts, "filename:*t*t")
639... so.write.string.nl(·fs, ts, [·DescFileName FROM 0 FOR DescFNameLen])
640... so.write.string(·fs, ts, "directory:*t*t")
641... so.write.string.nl(·fs, ts, [·OutDirName FROM 0 FOR OutDNameLen])
642:--- end of command line parsing-----
643
644... SEQ... cmnnpars-----
645... ParseCmdLine()
646
647... dfname IS [·DescFileName FROM 0 FOR DescFNameLen]:
648... odname IS [·OutDirName FROM 0 FOR OutDNameLen]:
649... SEQ
650... --- existence tests
651... so.open(·fs, ts, dfname, spt.text, spm.input, DescFileId, result)
652... condtest(·result <> spr.ok, "cannot open input file:", dfname)
653
654... --- description file parsing
655... PROC GetLine(·[BYTE line, INT len)-----
656... BYTE... result:
657... BOOL... more:
658... SEQ
659... more := TRUE
660... WHILE more
661... SEQ
662... so.gets(·fs, ts, DescFileId, len, line, result)
663... IF
664... result = spr.ok
665... more := (·len = 0) OR (·len > 0) AND

```



```

666 ..... ( (.line[0] = '#' ) OR ( .line[0] = '*s' ))
667 ..... TRUE
668 ..... len, more := 0, FALSE
669 ..... --- GetLine -----
670 .....
671 ..... PROC check( VAL BOOL success, VAL [ ] BYTE line, errmsg ) -----
672 ..... SEQ
673 ..... IF
674 ..... success
675 ..... SKIP
676 ..... TRUE
677 ..... SEQ
678 ..... so.write.string.nl( fs, ts, "error in description file" )
679 ..... so.write.string.nl( fs, ts, line )
680 ..... so.write.string.nl( fs, ts, "expected ->.." )
681 ..... so.write.string.nl( fs, ts, errmsg )
682 ..... so.exit( fs, ts, sps.failure )
683 ..... STOP
684 ..... --- check -----
685 .....
686 ..... BOOL, INT FUNCTION DescParseInt( VAL [ ] BYTE line, option ) -----
687 ..... INT number, ptr:
688 ..... BOOL ok:
689 ..... VALOF
690 ..... SEQ
691 ..... ptr, number, ok := SIZE option, 0, TRUE
692 ..... IF
693 ..... compare.strings( line, option ) = 1 ..... 1: option is a prefix of line
694 ..... SEQ
695 ..... next.int.from.line( line, ptr, number, ok )
696 ..... TRUE
697 ..... ok := FALSE
698 ..... RESULT ok, number
699 ..... --- DescParseInt -----
700 .....
701 ..... BOOL, INT, INT FUNCTION DescParseTxt( VAL [ ] BYTE line, option ) -----
702 ..... INT wstart, wlen, ptr:
703 ..... BYTE b:
704 ..... BOOL ok:
705 ..... [ MaxNameLen ] BYTE word:
706 ..... VALOF
707 ..... SEQ
708 ..... wstart, wlen, ok := SIZE option, 0, TRUE
709 ..... IF
710 ..... compare.strings( line, option ) = 1 ..... 1: option is a prefix of line
711 ..... SEQ
712 ..... wstart, b := search.no.match( "*s*t",
713 ..... [ line FROM SIZE option FOR ( ( SIZE line ) -- ( SIZE option ) ] )
714 ..... wstart := wstart + ( SIZE option )
715 ..... ptr := wstart
716 ..... next.word.from.line( line, ptr, wlen, word, ok )
717 ..... TRUE
718 ..... ok := FALSE
719 ..... RESULT ok, wstart, wlen
720 ..... --- DescParseTxt -----
721 .....
722 ..... BYTE result:
723 ..... BOOL ok:
724 ..... INT wstart, wlen:
725 ..... [ MaxNameLen ] BYTE line:
726 ..... SEQ
727 .....
728 ..... --- data file
729 ..... GetLine( line, len )
730 ..... aline IS [ line FROM 0 FOR len ]:
731 ..... SEQ
732 ..... ok, wstart, wlen := DescParseTxt( aline, "data:" )
733 ..... success := TRUE
734 ..... IF
735 ..... ok
736 ..... SEQ
737 ..... [ DataFileName FROM 0 FOR OutDNameLen ] := odname
738 ..... [ DataFileName FROM OutDNameLen FOR wlen ] := [ aline FROM wstart FOR wlen ]
739 ..... DataFNameLen := OutDNameLen + wlen
740 ..... so.test.exists( fs, ts, [ DataFileName FROM 0 FOR DataFNameLen ], success )

```

```

741 ..... TRUE
742 ..... success := FALSE
743 .....
744 ..... check( success, aline, "data: filename" )
745 ..... so.open( fs, ts, [ DataFileName FROM 0 FOR DataFNameLen ],
746 ..... spt.binary, spm.input, fid.data, result )
747 ..... so.write.string( fs, ts, "data file: *t*t" )
748 ..... so.write.string.nl( fs, ts, [ DataFileName FROM 0 FOR DataFNameLen ] )
749 ..... so.seek( fs, ts, fid.data, 0 ( INT32 ), spo.end, result )
750 ..... INT32 filelength:
751 ..... SEQ
752 ..... so.tell( fs, ts, fid.data, filelength, result )
753 ..... InitDataSize := INT filelength
754 ..... so.seek( fs, ts, fid.data, 0 ( INT32 ), spo.start, result )
755 .....
756 ..... --- input file / task
757 ..... GetLine( line, len )
758 ..... aline IS [ line FROM 0 FOR len ]:
759 ..... SEQ
760 ..... ok, wstart, wlen := DescParseTxt( aline, "input:" )
761 ..... success := TRUE
762 ..... IF
763 ..... ok
764 ..... SEQ
765 ..... [ TaskFileName FROM 0 FOR OutDNameLen ] := odname
766 ..... [ TaskFileName FROM OutDNameLen FOR wlen ] := [ aline FROM wstart FOR wlen ]
767 ..... TaskFNameLen := OutDNameLen + wlen
768 ..... so.test.exists( fs, ts, [ TaskFileName FROM 0 FOR TaskFNameLen ], success )
769 ..... TRUE
770 ..... success := FALSE
771 .....
772 ..... check( success, aline, "invalid file name or file not exists" )
773 ..... so.open( fs, ts, [ TaskFileName FROM 0 FOR TaskFNameLen ],
774 ..... spt.binary, spm.input, fid.task, result )
775 ..... so.write.string( fs, ts, "task file: *t*t" )
776 ..... so.write.string.nl( fs, ts, [ TaskFileName FROM 0 FOR TaskFNameLen ] )
777 ..... so.seek( fs, ts, fid.task, 0 ( INT32 ), spo.end, result )
778 ..... INT32 filelength:
779 ..... SEQ
780 ..... so.tell( fs, ts, fid.task, filelength, result )
781 ..... RootTaskSize := INT filelength
782 ..... so.seek( fs, ts, fid.task, 0 ( INT32 ), spo.start, result )
783 .....
784 .....
785 ..... PROC getsize( INT size, VAL [ ] BYTE name ) -----
786 ..... SEQ
787 ..... GetLine( line, len )
788 ..... aline IS [ line FROM 0 FOR len ]:
789 ..... BOOL success:
790 ..... [ 256 ] BYTE namebuf:
791 ..... [ ] BYTE namebuf IS [ namebuf FROM 0 FOR ( SIZE name ) + 1 ]:
792 ..... SEQ
793 ..... [ namebuf FROM 0 FOR ( SIZE name ) ] := name
794 ..... namebuf[ SIZE name ] := ':'
795 ..... success, size := DescParseInt( aline, namebuf )
796 ..... check( success, aline, namebuf )
797 ..... getsize -----
798 ..... SEQ
799 ..... getsize( ProfileSize, "profilesize" )
800 ..... getsize( EvalSize, "evalsize" )
801 ..... getsize( MaxTaskSize, "maxtasksize" )
802 :

```

B.3 Startup

```
commune.occ
```

```

803 #INCLUDE "config/commune.inc"
804
805 PROC commune( VAL INT ..... NodeId,
806 ..... [ ] CHAN OF INT inp, out, --- SIZE inp = SIZE out = NofNeighbours

```



```

807.....CHAN OF INT...envinp, envout,
808.....[]INT.....freememory)
809
810..VAL HEAD IS NodeId = 0:
811
812..#USE "obj/profile.tco"
813
814..INT...profBegMine, profBegNext:
815..INT...profUsed, profRecs:
816..INT...SafeSize:
817
818..SEQ
819...profSetup( NodeId, inp, out, envinp, envout,
820...profSize, profBegMine, profBegNext)
821
822...VAL profFlag IS ( profSize > 0 ):
823...-- ( profFlag = TRUE/FALSE ):: profiling enabled/disabled
824...SEQ
825...IF
826...profFlag
827...SafeSize := profSize
828...TRUE
829...SafeSize := 1
830...-- SIZE 0 would be invalid
831
832...profBuffer.....IS [ freememory FROM 0 FOR SafeSize ]:
833
834...-- cmnnData: rest of freememory
835...VAL cmnnDataSize IS ( SIZE freememory ) - SafeSize:
836...cmnnData.....IS [ freememory FROM SafeSize FOR cmnnDataSize ]:
837
838...SEQ
839...#INCLUDE "profile/profserv.inc"
840...[ 3 ]CHAN OF profProtocol profChan: -- magic numbers, oh
841...CHAN OF BOOL.....profStop:
842...PAR
843...PRI PAR
844...#USE "obj/profserv.tco"
845...profServer( profFlag, profChan, profStop,
846...profBuffer, profBegMine, profUsed, profRecs )
847...SKIP
848
849...SEQ
850...#USE "obj/cmnnmain.tco"
851...cmnnmain( NodeId, inp, out, envinp, envout,
852...profFlag, profChan, cmnnData )
853...profStop := TRUE
854
855...profClose( HEAD, inp, out, envinp, envout,
856...profFlag, profBegMine, profBegNext,
857...profBuffer, profUsed, profRecs )
858..IF
859...HEAD
860...envout := 0
861...TRUE
862...SKIP
863:

```

cmnnmain.occ

```

864-----
865-- cmnnmain.occ -- main phase, data load & processing
866-----
867#INCLUDE "config/commune.inc"
868#INCLUDE "profile/profserv.inc"
869#USE "debug.lib"
870
871PROC cmnnmain( VAL INT.....NodeId,
872...[]CHAN OF INT.....inp, out,
873...CHAN OF INT.....envinp, envout,
874...VAL BOOL.....profFlag,
875...[]CHAN OF profProtocol profChan,
876...[]INT.....freememory )
877
878..VAL HEAD IS NodeId = 0:

```

```

879..VAL TAIL IS NodeId = 1:
880..VAL NofNeighbours IS ( SIZE inp ):
881
882..INT evalSize, dataSize, maxtaskSize, roottaskSize:
883
884..INT i.n.t:
885..[]BYTE i.n.t RETYPES i.n.t:
886..VAL INTSIZE IS SIZE i.n.t:
887
888..PROC getSize()-----
889...SEQ
890...dout IS out[ HamBwSide ]:
891...dinp IS [ inp[ HamFwSide ], envinp ]:
892...dinp IS dinpv[ INT HEAD ]:
893...SEQ
894...-- task & init data size
895...dinp ? maxtaskSize
896...dinp ? roottaskSize
897...dinp ? evalSize
898...dinp ? dataSize
899...IF
900...TAIL
901...SKIP
902...TRUE
903...SEQ
904...dout := maxtaskSize
905...dout := roottaskSize
906...dout := evalSize
907...dout := dataSize
908...-- getSize-----
909
910..PROC getInitData( []INT buffer )-----
911...[]BYTE buffer RETYPES buffer:
912...dout IS out[ HamBwSide ]:
913...dinp IS [ inp[ HamFwSide ], envinp ]:
914...dinp IS dinpv[ INT HEAD ]:
915
916...CHAN OF BYTE dinp RETYPES dinp:
917...CHAN OF BYTE dout RETYPES dout:
918...BYTE d:
919
920...SEQ
921...SEQ i=0 FOR dataSize
922...SEQ
923...dinp ? d
924...buffer[ i ] := d
925...IF
926...TAIL
927...SKIP
928...TRUE
929...dout := d
930...-- getInitData-----
931
932..PROC sendResults( []INT buffer )-----
933...[]BYTE buffer RETYPES buffer:
934...doutv IS [ out[ HamBwSide ], envout ]:
935...dout IS doutv[ INT HEAD ]:
936...doutv IS doutv[ INT ( NOT HEAD ) ]:
937...dinp IS inp[ HamFwSide ]:
938
939...INT continue:
940...SEQ
941...IF
942...HEAD
943...doutx := 1
944...TRUE
945...SKIP
946...dinp ? continue
947
948...WHILE dataSize <> ( -1 )
949...SEQ
950...dout := dataSize
951
952...CHAN OF BYTE dout RETYPES dout:
953...SEQ i=0 FOR dataSize

```



```

954 ..... dout ! buffer[ i ]
955 .....
956 ..... dataSize := continue
957 ..... IF
958 ..... dataSize >= 0
959 ..... SEQ
960 ..... CHAN OF BYTE dinp RETYPES dinp:
961 ..... SEQ i=0 FOR dataSize
962 ..... dinp ? buffer[ i ]
963 ..... dinp ? continue
964 ..... TRUE
965 ..... SKIP
966 .....
967 ..... dout ! dataSize
968 ..... sendResults -----
969 .....
970 .. SEQ
971 .... getSize()
972 .....
973 .... profChan[ 0 ] ! EVENT.MARK; maxtaskSize
974 .... profChan[ 0 ] ! EVENT.MARK; roottaskSize
975 .... profChan[ 0 ] ! EVENT.MARK; evalSize
976 .... profChan[ 0 ] ! EVENT.MARK; dataSize
977 .....
978 .... VAL maxtaskSizeINT IS ( maxtaskSize / INTSIZE ) + 1;
979 .... VAL NofGuardBuffers IS 3: -- inp1, inp2, out; see transfer
980 .... VAL GuardSizeINT IS maxtaskSizeINT * NofGuardBuffers:
981 .... VAL sentinelBufSizeINT IS NofNeighbours * GuardSizeINT:
982 .... VAL freememSize IS ( SIZE.freememory ) -- sentinelBufSizeINT:
983 .... sentinelBuffer IS [ freememory FROM 0 FOR sentinelBufSizeINT ]:
984 .... freememory IS [ freememory FROM sentinelBufSizeINT FOR freememSize ]:
985 .....
986 .... VAL evalSizeINT IS ( evalSize / INTSIZE ) + 1;
987 .... VAL storeSizeINT IS ( SIZE.freememory ) -- evalSizeINT:
988 .... bossBuffer IS [ freememory FROM 0 FOR evalSizeINT ]:
989 .... storememory IS [ freememory FROM evalSizeINT FOR storeSizeINT ]:
990 .....
991 .... SEQ
992 .... getInitData( bossBuffer )
993 .....
994 .... #INCLUDE "boss/cmmnboss.inc"
995 .... #INCLUDE "store/cmmnstor.inc"
996 .... #USE "obj/storifac.tco"
997 .....
998 ..... [ MaxNofNeighbours ] CHAN OF commandProtocol com:
999 ..... com IS [ com FROM 0 FOR NofNeighbours ]: -- boss to guard
1000 ..... [ MaxNofNeighbours ] CHAN OF BOOL ..... ack:
1001 ..... ack IS [ ack FROM 0 FOR NofNeighbours ]: -- guard to boss
1002 .....
1003 ..... [ MaxNofNeighbours ] CHAN OF storeReqProtocol reqv:
1004 ..... reqv IS [ reqv FROM 0 FOR NofNeighbours + 1 ]:
1005 ..... [ MaxNofNeighbours ] CHAN OF storeAnsProtocol ansv:
1006 ..... ans IS [ ansv FROM 0 FOR NofNeighbours + 1 ]:
1007 .....
1008 ..... CHAN OF BOOL stopStore:
1009 .....
1010 ..... breq IS req[ 0 ]: -- for boss
1011 ..... bans IS ans[ 0 ]:
1012 ..... sreq IS [ req FROM 1 FOR NofNeighbours ]: -- for sentinel
1013 ..... sans IS [ ans FROM 1 FOR NofNeighbours ]:
1014 .....
1015 ..... PROC getRoot() -----
1016 ..... CHAN OF BYTE envinp RETYPES envinp:
1017 ..... [ ] BYTE data RETYPES sentinelBuffer:
1018 ..... IF
1019 ..... NOT HEAD
1020 ..... SKIP
1021 ..... TRUE -- task in
1022 ..... SEQ
1023 ..... SEQ i=0 FOR roottaskSize
1024 ..... envinp ? data[ i ]
1025 .....
1026 ..... INT booked:
1027 ..... SEQ
1028 ..... bookput( breq, bans, type.mass, 1, booked)

```

```

1029 ..... DEBUG.ASSERT( booked = 1 )
1030 ..... push( breq, bans, type.mass, [ data FROM 0 FOR roottaskSize ] )
1031 ..... getRoot -----
1032 .....
1033 ..... PAR
1034 ..... PRI PAR
1035 ..... #USE "obj/cmmnstor.tco"
1036 ..... store( req, ans, stopStore, profChan[ 0 ], maxtaskSizeINT, storememory )
1037 ..... SKIP
1038 .....
1039 ..... SEQ
1040 ..... getRoot()
1041 .....
1042 ..... PRI PAR
1043 ..... #USE "obj/sentinel.tco"
1044 ..... sentinel( NodeId, inp, out, com, ack, sreq, sans,
1045 ..... sentinelBuffer, NofGuardBuffers, maxtaskSizeINT )
1046 .....
1047 ..... SEQ
1048 ..... #USE "obj/cmmnboss.tco"
1049 ..... cmmnboss( NodeId, com, ack, breq, bans,
1050 ..... [ profChan FROM 1 FOR 2 ], bossBuffer, dataSize, maxtaskSizeINT )
1051 .....
1052 ..... --
1053 ..... stopStore ! TRUE
1054 .....
1055 ..... INT dummy:
1056 ..... IF
1057 ..... HEAD
1058 ..... SEQ
1059 ..... out [ HamFwSide ] ! dummy
1060 ..... inp [ HamBwSide ] ? dummy
1061 ..... TRUE
1062 ..... SEQ
1063 ..... inp [ HamBwSide ] ? dummy
1064 ..... out [ HamFwSide ] ! dummy
1065 .....
1066 ..... IF
1067 ..... HEAD
1068 ..... envout ! 0
1069 ..... TRUE
1070 ..... SKIP
1071 .....
1072 ..... sendResults( bossBuffer )
1073 ..... -----

```

B.4 Profiler

```
profile/profdiff.occ
```

```

1074 PROC profDiff( VAL BOOL ..... HEAD,
1075 ..... CHAN OF INT inp, out,
1076 ..... INT ..... TimeMine, TimeNext )
1077 .....
1078 .. INLINE PROC profDiffActive()
1079 ..... INT ..... TimeNow:
1080 ..... TIMER Clock:
1081 ..... SEQ
1082 ..... out ! 0 ..... dummy zero for synchronisation
1083 ..... Clock ? TimeNow ..... other side is doing the same
1084 ..... out ! TimeNow
1085 .....
1086 .....
1087 .. INLINE PROC profDiffPassiv()
1088 ..... TIMER Clock:
1089 ..... SEQ
1090 ..... INT dummy:
1091 ..... inp ? dummy ..... synchronisation
1092 ..... Clock ? TimeMine ..... other side is doing the same
1093 ..... inp ? TimeNext
1094 .....

```



```

1095
1096 .. PRI· PAR
1097 .. SEQ
1098 .. IF
1099 .. HEAD
1100 .. SEQ
1101 .. profDiffActive()
1102 .. profDiffPassiv()
1103 .. TRUE
1104 .. SEQ
1105 .. profDiffPassiv()
1106 .. profDiffActive()
1107 .. SKIP
1108 :-- profDiff -----

```

profile/profile.occ

```

1109 #INCLUDE "config/commune.inc"
1110 #USE "obj/profdiff.tco"
1111
1112 -----
1113 PROC profSetup(· VAL· INT· ..... NodeId,
1114 .. [·]CHAN· OF· INT· inp, out,
1115 .. CHAN· OF· INT· envinp, envout,
1116 .. INT· ..... ProfSize,
1117 .. INT· ..... BegMine, BegNext)
1118
1119 .. VAL· HEAD· IS· NodeId· =· 0:
1120
1121 .. PROTOCOL· HAMILTON· IS· INT;· INT:
1122
1123 .. PROC· profFlagDistrib() -----
1124 .. CHAN· OF· HAMILTON· distrout· RETYPES· out[· HamFwSide]:
1125 .. CHAN· OF· HAMILTON· distrinp· RETYPES· inp[· HamBwSide]:
1126
1127 .. INT· NodeIdChk:
1128
1129 .. INLINE· PROC· datainp() -----
1130 .. distrinp· ?· NodeIdChk;· ProfSize
1131 .. :-- datainp -----
1132
1133 .. INLINE· PROC· dataout() -----
1134 .. SEQ
1135 .. IF
1136 .. NodeIdChk· <· NodeId
1137 .. NodeIdChk· :=· -1· ..... not· a· valid· Node· id
1138 .. TRUE
1139 .. NodeIdChk· :=· NodeId+1
1140 .. distrout· !· NodeIdChk;· ProfSize
1141 .. :-- dataout -----
1142
1143 .. IF
1144 .. HEAD
1145 .. SEQ
1146 .. envinp· ?· ProfSize
1147 .. NodeIdChk· :=· NodeId
1148 .. dataout()
1149 .. datainp()
1150 .. envout· !· NodeIdChk
1151 .. TRUE
1152 .. SEQ
1153 .. datainp()
1154 .. dataout()
1155 .. :-- profFlagDistrib -----
1156
1157 .. SEQ
1158 .. profFlagDistrib()
1159 .. profDiff(· HEAD, inp[· HamFwSide], out[· HamBwSide], BegMine, BegNext)
1160
1161 :-- profSetup -----
1162
1163 -----
1164 PROC profClose(· VAL· BOOL· ..... HEAD,
1165 .. [·]CHAN· OF· INT· inp, out,
1166 .. CHAN· OF· INT· envinp, envout,

```

```

1167 .. VAL· BOOL· ..... profFlag,
1168 .. VAL· INT· ..... BegMine, BegNext,
1169 .. [·]INT· ..... profBuffer,
1170 .. VAL· INT· ..... profUsed, profRecs)
1171
1172 .. INT· tBegMine, tBegNext:
1173 .. INT· tEndMine, tEndNext:
1174 .. INT· tUsed, tRecs:
1175
1176 .. PROC· profCollect()
1177 .. cinp· IS· inp[· HamFwSide]:
1178 .. coutv· IS· [· out[· HamBwSide], envout]:
1179 .. cout· IS· coutv[· INT· HEAD]:
1180 .. hout· IS· coutv[· INT· (· NOT· HEAD)]:
1181
1182 .. PROC· send() -----
1183 .. SEQ
1184 .. cout· !· 1· --- data· follows
1185 .. cout· !· tBegMine
1186 .. cout· !· tBegNext
1187 .. cout· !· tEndMine
1188 .. cout· !· tEndNext
1189 .. cout· !· tRecs
1190 .. cout· !· tUsed
1191 .. SEQ· i=0· FOR· tUsed
1192 .. cout· !· profBuffer[· i]
1193 .. :-- send -----
1194
1195 .. INT· continue:
1196 .. SEQ
1197 .. :-- profile· data· collection
1198 .. tBegMine, tBegNext, tUsed, tRecs· :=· BegMine, BegNext, profUsed, profRecs
1199
1200 .. IF
1201 .. HEAD
1202 .. hout· !· 0· --- no· more· data
1203 .. TRUE
1204 .. SKIP
1205
1206 .. cinp· ?· continue
1207 .. send()
1208
1209 .. WHILE· continue· =· 1
1210 .. SEQ
1211 .. cinp· ?· tBegMine
1212 .. cinp· ?· tBegNext
1213 .. cinp· ?· tEndMine
1214 .. cinp· ?· tEndNext
1215 .. cinp· ?· tRecs
1216 .. cinp· ?· tUsed
1217 .. SEQ· i=0· FOR· tUsed
1218 .. cinp· ?· profBuffer[· i]
1219
1220 .. send()
1221 .. cinp· ?· continue
1222
1223 .. cout· !· 0· --- no· more· data
1224 .. :-- profCollect -----
1225
1226 .. SEQ
1227 .. IF
1228 .. profFlag
1229 .. SEQ
1230 .. profDiff(· HEAD, inp[· HamFwSide], out[· HamBwSide], tEndMine, tEndNext)
1231 .. profCollect()
1232 .. TRUE
1233 .. SKIP
1234 :-- profClose -----

```

profile/profserv.inc

```

1235 PROTOCOL· profProtocol· IS· BYTE;· INT· :-- tag;· value
1236
1237 :-- tag· values
1238 VAL· EVENT· START· ..... IS· 0· (· BYTE);· :-- value=time

```



```

1239 VAL EVENT.END.....IS 1. (. BYTE) :-- value=time
1240 VAL EVENT.MARK.....IS 2. (. BYTE) :-- value=...
1241 VAL EVENT.MASS.....IS 3. (. BYTE) :-- value=noftasks

```

```
profile/profserv.occ
```

```

1242 #INCLUDE "profile/profserv.inc"
1243
1244 PROC profServer(. VAL BOOL..... profFlag,
1245 ..... [CHAN OF profProtocol.inp,
1246 ..... CHAN OF BOOL..... profStop,
1247 ..... [INT..... profBuffer,
1248 ..... VAL INT..... DefaultLast,
1249 ..... INT nofSlots, nofRecords)
1250
1251 . INT INLINE FUNCTION msbpos(. VAL INT x) :-----
1252 . VAL mask IS [ #FFFF0000, #FF00FF00, #F0F0F0F0, #CCCCCCCC, #AAAAAAA] :
1253 . INT pos :
1254 . VAL OF
1255 . INT n, y :
1256 . SEQ
1257 . n, pos := x, 0
1258 . SEQ i=0 FOR SIZE mask
1259 . SEQ
1260 . pos, y := pos << 1, n.BITAND mask[i]
1261 . IF
1262 . y = 0
1263 . SKIP
1264 . TRUE
1265 . n, pos := y, pos.BITOR 1
1266 . RESULT pos + (. INT (x <> 0))
1267 . msbpos :-----
1268
1269 . INT slotLen :
1270 . INT slot :
1271 . BOOL bufFull :
1272 . INT LastEvent :
1273 . TIMER clock :
1274
1275 . INLINE PROC profFlush() :-----
1276 . SEQ
1277 . profBuffer[. nofSlots] := slot >> (. 32-slotLen)
1278 . nofSlots := nofSlots+1
1279 . profFlush :-----
1280
1281 . INLINE PROC RecordEvent(. VAL BYTE tag, VAL INT value) :-----
1282
1283 . VAL eraser IS [ #00000000, #80000000, #C0000000, #E0000000,
1284 . #F0000000, #F8000000, #FC000000, #FE000000,
1285 . #FF000000, #FF800000, #FFC00000, #FFE00000,
1286 . #FFF00000, #FFF80000, #FFFC0000, #FFFE0000,
1287 . #FFFF0000, #FFFF8000, #FFFFC000, #FFFFE000,
1288 . #FFFFFF00, #FFFFFF80, #FFFFFFC0, #FFFFFFE0,
1289 . #FFFFFFF0, #FFFFFFF8, #FFFFFFFC, #FFFFFFFE] :
1290
1291 . INT free :
1292 . SEQ
1293 . free := 32-slotLen
1294 . VAL overflow IS len-free :
1295 . VAL xhigh IS x << (. 32-len) :
1296 . IF
1297 . overflow < 0
1298 . SEQ
1299 . slot := (. slot >> len).BITOR xhigh
1300 . slotLen := slotLen+len
1301 . TRUE
1302 . SEQ
1303 . profBuffer[. nofSlots] := (. slot >> free).BITOR (. xhigh << overflow)
1304 . nofSlots, slotLen := nofSlots+1, overflow
1305 . slot := xhigh.BITAND eraser[slotLen]
1306
1307 . putbits :-----
1308
1309 . IF
1310

```

```

1311 ..... (. NOT profFlag).OR bufFull
1312 ..... SKIP
1313 ..... TRUE
1314 ..... SEQ
1315 ..... nofRecords := nofRecords+1
1316 ..... time :-----
1317 ..... INT diff, msbp.abs, msbp.rel :
1318 ..... INT now :
1319 ..... SEQ
1320 ..... clock ? now
1321 ..... diff := now MINUS LastEvent
1322 ..... msbp.abs, msbp.rel := msbpos(. now), msbpos(. diff)
1323 ..... IF
1324 ..... msbp.abs <= msbp.rel
1325 ..... SEQ
1326 ..... putbits(. 0, 1) :----- a single 0 : absolute time
1327 ..... putbits(. msbp.abs, 5) :----- 5 bits : next field len
1328 ..... putbits(. now, msbp.abs) :----- field data
1329 ..... TRUE
1330 ..... SEQ
1331 ..... putbits(. 1, 1) :----- a single 1 : relative time
1332 ..... putbits(. msbp.rel, 5) :----- 5 bits : next field len
1333 ..... putbits(. diff, msbp.rel) :----- field data
1334 ..... LastEvent := now
1335 ..... tag :-----
1336 ..... INT msbp.tag :
1337 ..... SEQ
1338 ..... msbp.tag := msbpos(. INT tag)
1339 ..... putbits(. msbp.tag, 3) :----- 3 bits : field len
1340 ..... putbits(. INT tag, msbp.tag) :----- field data
1341 ..... value :-----
1342 ..... INT msbp.val :
1343 ..... SEQ
1344 ..... msbp.val := msbpos(. value)
1345 ..... putbits(. msbp.val, 5) :----- 5 bits : field len
1346 ..... putbits(. value, msbp.val) :----- field data
1347 ..... bufFull := nofSlots > (. (. SIZE profBuffer) - 10) : play safe
1348 ..... bufFull := nofSlots > (. (. SIZE profBuffer) - 10) : play safe
1349
1350 . RecordEvent :-----
1351
1352 . BOOL quit :
1353 . BYTE tag :
1354 . INT value :
1355 . SEQ
1356 . nofSlots, nofRecords, slotLen, bufFull := 0, 0, 0, FALSE
1357
1358 . LastEvent := DefaultLast
1359 . RecordEvent(. EVENT.START, SIZE profBuffer)
1360
1361 . quit := FALSE
1362 . WHILE NOT quit
1363 . ALT
1364 . profStop ? quit
1365 . quit := TRUE
1366 . ALT i=0 FOR SIZE inp
1367 . inp[i] ? tag, value
1368 . RecordEvent(. tag, value)
1369
1370 . value := nofRecords+1
1371 . RecordEvent(. EVENT.END, value)
1372
1373 . profFlush()
1374 . SEQ

```

B.5 Store

```
store/cmmnstor.inc
```

```

1375 PROTOCOL storeReqProtocol :-----
1376 . CASE

```



```

1377 ---- any type
1378 ---- sr.wait
1379 ---- sr.get; INT type
1380 ---- sr.put; INT type -- preparation
1381 ---- sr.putdata; INT::[]BYTE data
1382 ---- mass related
1383 ---- srm.bookget; INT amount
1384 ---- srm.bookput; INT amount
1385 ---- srm.cancel
1386 ---- srm.close
1387 ---- storeRequestProtocol -----
1388
1389 PROTOCOL storeAnsProtocol -----
1390 -- CASE
1391 ---- any type
1392 ---- sa.wait
1393 ---- sa.get; INT::[]BYTE data
1394 ---- mass related
1395 ---- sam.bookget; INT amount available
1396 ---- sam.bookput; INT amount available
1397 ----
1398 ---- sa.ready ----- for debugging
1399 ---- storeAnswerProtocol -----
1400
1401 -- direction constants for mass data
1402 VAL dir.send IS 0:
1403 VAL dir.recv IS 1:
1404
1405 -- type indices -----
1406
1407 VAL type.mass IS 0: -- mass data
1408
1409 -- positive values denote proper control data types
1410 VAL type.stat IS 1: -- mass data statistics
1411 VAL type.token IS 2: -- termination token
1412 VAL type.wtoken IS 3: -- token to wait for
1413 VAL type.global IS 4: -- global (sort of) data
1414
1415 BOOL INLINE FUNCTION is.mass (VAL INT type) IS (type = type.mass):
1416 BOOL INLINE FUNCTION is.ctrl (VAL INT type) IS (type > type.mass):
1417 -- type indices -----
1418
1419 ----
1420 -- mass statistics -----
1421
1422 -- operations
1423 -- get gives a 3-element vector
1424 ---- 0: -- nof used slots
1425 ---- 1: -- nof free slots
1426 ---- 2: -- total nof slots (capacity of store)
1427 -- put ignored
1428
1429 ----
1430 -- token -----
1431
1432 -- token colors
1433 VAL token.invisible IS 0 (BYTE): -- meaning no token here, for init
1434 -- positive values mean store is empty
1435 -- negative values mean store is full
1436
1437 -- token operations
1438 -- get gives back the current color and sets token.invisible as new color
1439 -- put sets the token to the color given
1440
1441 ----
1442 -- global -----
1443
1444 -- not implemented yet

```

store/cmmnstor.occ

```

1445 #INCLUDE "config/commune.inc"
1446 #INCLUDE "profile/profserv.inc"
1447 #INCLUDE "store/cmmnstor.inc"
1448 #USE "debug.lib"

```

```

1449
1450 VAL MaxNofClients IS MaxNofNeighbours+1: -- + boss
1451
1452 PROC store ( [CHAN OF storeReqProtocol req,
1453 [CHAN OF storeAnsProtocol ans,
1454 CHAN OF BOOL stop,
1455 CHAN OF profProtocol profChan,
1456 VAL INT maxSize,
1457 [INT memory)
1458
1459 -- VAL NofClients IS (SIZE req):
1460
1461 -- BYTE token:
1462 -- BOOL token.wait:
1463 -- INT token.clt:
1464
1465 -- VAL slotSize IS maxSize+2: -- +2: link field, actual size
1466 -- VAL nofSlots IS (SIZE memory) / slotSize:
1467
1468 -- INT nofUsed, nofFree:
1469
1470 -- INT firstUsed, firstFree:
1471 -- VAL NIL IS -1:
1472
1473 -- BOOL waiting:
1474
1475 -- per client data -----
1476
1477 -- waiting flag
1478 -- [MaxNofClients]BOOL waitFlag:
1479 -- waitFlag IS [waitFlag FROM 0 FOR NofClients]:
1480
1481 -- pointer to booked slot chain
1482 -- [MaxNofClients]INT slots:
1483 -- slots IS [slots FROM 0 FOR NofClients]:
1484
1485 -- indicates where slots were booked from (needed for cancel, close)
1486 -- [MaxNofClients]BYTE bookFlag:
1487 -- bookFlag IS [bookFlag FROM 0 FOR NofClients]:
1488
1489 -- constants for bookFlag
1490 -- VAL booked.from.used IS 0 (BYTE):
1491 -- VAL booked.from.free IS 1 (BYTE):
1492
1493
1494 -- BOOL FUNCTION valididx (VAL INT idx) IS (idx >= 0) AND (idx < nofSlots):
1495
1496 -- PROC setlink (VAL INT idx, ptr) -----
1497 -- SEQ
1498 -- DEBUG ASSERT (valididx (idx))
1499 -- DEBUG ASSERT (valididx (ptr) OR (ptr = NIL))
1500 -- slotlink IS memory [idx*slotSize]:
1501 -- slotlink := ptr
1502 -- setlink -----
1503
1504 -- INT FUNCTION getlink (VAL INT idx) IS memory [idx*slotSize]:
1505
1506 -- PROC setsize (VAL INT idx, size) -----
1507 -- SEQ
1508 -- DEBUG ASSERT (valididx (idx))
1509 -- slotsize IS memory [(idx*slotSize)+1]:
1510 -- slotsize := size
1511 -- setsize -----
1512
1513 -- INT FUNCTION getsize (VAL INT idx) IS memory [(idx*slotSize)+1]:
1514
1515 -- PROC setdata (VAL INT idx, [BYTE data) ---
1516 -- SEQ
1517 -- DEBUG ASSERT (valididx (idx))
1518 -- DEBUG ASSERT (maxSize >= (SIZE data))
1519 -- setsize (idx, SIZE data)
1520
1521 -- slotdata IS [memory FROM (idx*slotSize)+2 FOR maxSize]:
1522 -- [BYTE slotdata RETYPES slotdata:
1523 -- [slotdata FROM 0 FOR SIZE data] := data

```



```

1524 ..-- setdata -----
1525
1526 ..PROC getdata( VAL INT idx, INT size, [] BYTE data) ---
1527 ..SEQ
1528 ..DEBUG.ASSERT( valididx( idx ))
1529 ..size := getsize( idx )
1530
1531 ..slotdata IS [ memory FROM ( idx*slotSize)+2 FOR maxSize ]:
1532 ..[] BYTE slotdata RETYPES slotdata:
1533 ..[ data FROM 0 FOR size ] := [ slotdata FROM 0 FOR size ]
1534 ..-- getdata -----
1535
1536 ..PROC init() -----
1537 ..SEQ
1538 ..DEBUG.ASSERT( NofClients <= MaxNofClients )
1539
1540 ..waiting := FALSE
1541
1542 ..SEQ i=0 FOR NofClients
1543 ..slots[ i ], waitFlag[ i ] := NIL, FALSE
1544
1545 ..-- mass -----
1546 ..-- initial free chain
1547 ..SEQ i=0 FOR nofSlots-1
1548 ..setlink( i, i+1 )
1549 ..setlink( nofSlots-1, NIL )
1550
1551 ..nofUsed, firstUsed := 0, ..... NIL
1552 ..nofFree, firstFree := nofSlots, 0
1553
1554 ..-- ctrl -----
1555 ..-- token
1556 ..token.wait := FALSE
1557 ..token := token.invisible
1558
1559 ..profChan ! EVENT.MASS; nofUsed
1560 ..init -----
1561
1562 ..PROC waker() -----
1563 ..SEQ
1564 ..SEQ i=0 FOR NofClients
1565 ..IF
1566 ..waitFlag[ i ]
1567 ..SEQ
1568 ..ans[ i ] ! sa.wait
1569 ..waitFlag[ i ] := FALSE
1570 ..TRUE
1571 ..SKIP
1572 ..waiting := FALSE
1573 ..waker -----
1574
1575 ..PROC token.get( VAL INT clt ) -----
1576 ..SEQ
1577 ..ans[ clt ] ! sa.get; 1: [ token ]
1578 ..token := token.invisible
1579 ..token.wait := FALSE
1580 ..ctrl.get -----
1581
1582 ..PROC get( VAL INT clt, type )
1583
1584 ..PROC mass.get() -----
1585 ..VAL idx IS slots[ clt ]:
1586 ..SEQ
1587 ..DEBUG.ASSERT( idx <> NIL )
1588 ..slotsize IS memory[ ( idx*slotSize)+1 ]:
1589 ..[] BYTE slotdata RETYPES [ memory FROM ( idx*slotSize)+2 FOR maxSize ]:
1590 ..ans[ clt ] ! sa.get; slotsize::slotdata
1591 ..mass.get -----
1592
1593 ..PROC stat.get() -----
1594 ..[ 3 ] INT statvec:
1595 ..SEQ
1596 ..statvec[ 0 ], statvec[ 1 ], statvec[ 2 ] := nofUsed, nofFree, nofSlots
1597 ..[] BYTE statvec RETYPES statvec:
1598 ..ans[ clt ] ! sa.get; ( SIZE statvec )::statvec

```

```

1599 ..-- stat.get -----
1600
1601 ..CASE type
1602 ..type.mass
1603 ..mass.get()
1604 ..type.stat
1605 ..stat.get()
1606 ..type.token
1607 ..token.get( clt )
1608 ..type.wtoken
1609 ..IF
1610 ..token := token.invisible
1611 ..token.wait, token.clc := TRUE, clt
1612 ..TRUE
1613 ..token.get( clt )
1614 ..get -----
1615
1616 ..PROC put( VAL INT clt, type )
1617
1618 ..PROC mass.put() -----
1619 ..VAL idx IS slots[ clt ]:
1620 ..SEQ
1621 ..DEBUG.ASSERT( idx <> NIL )
1622 ..slotsize IS memory[ ( idx*slotSize)+1 ]:
1623 ..[] BYTE slotdata RETYPES [ memory FROM ( idx*slotSize)+2 FOR maxSize ]:
1624 ..req[ clt ] ? CASE sr.putdata; slotsize::slotdata
1625 ..mass.put -----
1626
1627 ..PROC token.put() -----
1628 ..INT one:
1629 ..[ 1 ] BYTE token RETYPES token:
1630 ..SEQ
1631 ..req[ clt ] ? CASE sr.putdata; one::token
1632 ..DEBUG.ASSERT( one = 1 )
1633 ..token.put -----
1634
1635 ..CASE type
1636 ..type.mass
1637 ..mass.put()
1638 ..type.stat
1639 ..SKIP
1640 ..type.token
1641 ..SEQ
1642 ..token.put()
1643 ..IF
1644 ..token.wait
1645 ..token.get( token.clc )
1646 ..TRUE
1647 ..SKIP
1648 ..put -----
1649
1650 ..PROC relocate( INT base, source ) -----
1651 ..INT tmp:
1652 ..SEQ
1653 ..tmp := source
1654 ..source := getlink( source )
1655 ..setlink( tmp, base )
1656 ..base := tmp
1657 ..relocate -----
1658
1659 ..PROC backtoUsed( VAL INT clt ) -----
1660 ..SEQ
1661 ..relocate( firstUsed, slots[ clt ] )
1662 ..nofUsed := nofUsed+1
1663 ..backtoUsed -----
1664
1665 ..PROC backtoFree( VAL INT clt ) -----
1666 ..SEQ
1667 ..relocate( firstFree, slots[ clt ] )
1668 ..nofFree := nofFree+1
1669 ..backtoFree -----
1670
1671 ..PROC cancel( VAL INT clt ) -----
1672 ..SEQ
1673 ..DEBUG.ASSERT( slots[ clt ] <> NIL )

```



```

1674 ..... CASE bookFlag[. clt]
1675 ..... booked.from.used
1676 ..... backtoUsed(. clt)
1677 ..... booked.from.free
1678 ..... backtoFree(. clt)
1679 ..... cancel
1680 ..... -----
1681 .. PROC closeall(. VAL INT clt) -----
1682 ..... SEQ
1683 ..... WHILE slots[. clt] <> NIL
1684 ..... SEQ
1685 ..... CASE bookFlag[. clt]
1686 ..... booked.from.used
1687 ..... backtoFree(. clt)
1688 ..... booked.from.free
1689 ..... backtoUsed(. clt)
1690 ..... profChan ! EVENT.MASS; .nofUsed
1691 ..... closeall
1692 ..... -----
1693 .. PROC close(. VAL INT clt) -----
1694 ..... SEQ
1695 ..... DEBUG.ASSERT(. slots[. clt] <> NIL)
1696 ..... CASE bookFlag[. clt]
1697 ..... booked.from.used
1698 ..... backtoFree(. clt)
1699 ..... booked.from.free
1700 ..... backtoUsed(. clt)
1701 ..... profChan ! EVENT.MASS; .nofUsed
1702 ..... close
1703 ..... -----
1704 .. PROC detach(. INT base, . VAL INT amount) -----
1705 ..... INT next:
1706 ..... SEQ
1707 ..... next := base
1708 ..... SEQ i=0 FOR amount-1
1709 ..... next := getlink(. next)
1710 ..... base := getlink(. next)
1711 ..... setlink(. next, NIL)
1712 ..... detach
1713 ..... -----
1714 .. PROC bookget(. VAL INT clt, . amount) -----
1715 ..... SEQ
1716 ..... IF
1717 ..... nofUsed >= amount --- are there enough slots to book?
1718 ..... SEQ
1719 ..... IF
1720 ..... slots[. clt] <> NIL
1721 ..... closeall(. clt)
1722 ..... TRUE
1723 ..... SKIP
1724 ..... ans[. clt] ! sam.bookget; amount
1725 ..... slots[. clt], nofUsed := firstUsed, nofUsed-amount
1726 ..... bookFlag[. clt] := booked.from.used
1727 ..... detach(. firstUsed, amount)
1728 ..... TRUE
1729 ..... ans[. clt] ! sam.bookget; nofUsed
1730 ..... bookget
1731 ..... -----
1732 .. PROC bookput(. VAL INT clt, . amount) -----
1733 ..... SEQ
1734 ..... IF
1735 ..... amount <= nofFree --- are there enough empty slots?
1736 ..... SEQ
1737 ..... IF
1738 ..... slots[. clt] <> NIL
1739 ..... closeall(. clt)
1740 ..... TRUE
1741 ..... SKIP
1742 ..... ans[. clt] ! sam.bookput; amount
1743 ..... slots[. clt], nofFree := firstFree, nofFree-amount
1744 ..... bookFlag[. clt] := booked.from.free

```

```

1749 ..... detach(. firstFree, . amount)
1750 ..... TRUE
1751 ..... ans[. clt] ! sam.bookput; .nofFree
1752 ..... bookput
1753 ..... -----
1754 .. BOOL quit, . callwaker:
1755 ..... -----
1756 .. PROC dispatcher() -----
1757 ..... ALT
1758 ..... stop ? quit
1759 ..... quit := TRUE
1760 ..... INT size, . ctrlidx, . type:
1761 ..... PRI ALT clt=0 FOR NofClients
1762 ..... req[. clt] ? CASE
1763 ..... sr.wait
1764 ..... callwaker, . waiting, . waitFlag[. clt] := FALSE, TRUE, TRUE
1765 ..... sr.get; . type
1766 ..... get(. clt, . type)
1767 ..... sr.put; . type
1768 ..... put(. clt, . type)
1769 ..... INT amount:
1770 ..... srm.bookget; . amount
1771 ..... SEQ
1772 ..... DEBUG.ASSERT(. amount > 0)
1773 ..... bookget(. clt, . amount)
1774 ..... INT amount:
1775 ..... srm.bookput; . amount
1776 ..... SEQ
1777 ..... DEBUG.ASSERT(. amount > 0)
1778 ..... bookput(. clt, . amount)
1779 ..... srm.cancel
1780 ..... cancel(. clt)
1781 ..... srm.close
1782 ..... close(. clt)
1783 ..... -----
1784 .. dispatcher -----
1785 ..... SEQ
1786 ..... store
1787 ..... init()
1788 ..... quit := FALSE
1789 ..... -----
1790 .. main loop -----
1791 ..... WHILE NOT quit
1792 ..... SEQ
1793 ..... callwaker := TRUE
1794 ..... dispatcher()
1795 ..... IF
1796 ..... callwaker AND waiting
1797 ..... waker()
1798 ..... TRUE
1799 ..... SKIP
1800 ..... store

```

store/storifac.occ

```

1808 #INCLUDE "store/cmmnstor.inc"
1809 #USE "debug.lib"
1810 ..... PROC bookget(. CHAN OF storeReqProtocol req,
1811 ..... CHAN OF storeAnsProtocol ans,
1812 ..... VAL INT storeAnsProtocol type,
1813 ..... VAL INT amount,
1814 ..... INT avail)
1815 ..... CASE type
1816 ..... type mass
1817 ..... SEQ
1818 ..... req ! srm.bookget; amount
1819 ..... ans ? CASE sam.bookget; avail
1820 .....

```



```

1821 ..... ELSE
1822 ..... SEQ
1823 ..... DEBUG.ASSERT(. amount = 1)
1824 ..... avail := amount
1825 ..... bookget
1826 -----
1827 PROC bookput(. CHAN OF storeReqProtocol req,
1828 ..... CHAN OF storeAnsProtocol ans,
1829 ..... VAL INT ..... type,
1830 ..... VAL INT ..... amount,
1831 ..... INT ..... avail)
1832 .. CASE type
1833 .. type.mass
1834 .. SEQ
1835 ..... req ! srm.bookput; amount
1836 ..... ans ? CASE sam.bookput; avail
1837 ..... ELSE
1838 ..... SEQ
1839 ..... DEBUG.ASSERT(. amount = 1)
1840 ..... avail := amount
1841 ..... bookput
1842 -----
1843 PROC book(. CHAN OF storeReqProtocol req,
1844 ..... CHAN OF storeAnsProtocol ans,
1845 ..... VAL INT ..... type,
1846 ..... VAL INT ..... dir,
1847 ..... VAL INT ..... amount,
1848 ..... INT ..... avail)
1849 .. CASE dir
1850 .. dir.send
1851 ..... bookget(. req, ans, type, amount, avail)
1852 .. dir.recv
1853 ..... bookput(. req, ans, type, amount, avail)
1854 ..... book
1855 -----
1856 PROC get(. CHAN OF storeReqProtocol req,
1857 ..... CHAN OF storeAnsProtocol ans,
1858 ..... VAL INT ..... type,
1859 ..... INT ..... size,
1860 ..... [] BYTE ..... data)
1861 .. SEQ
1862 ..... req ! sr.get; type
1863 ..... ans ? CASE sa.get; size::data
1864 ..... get
1865 -----
1866 PROC put(. CHAN OF storeReqProtocol req,
1867 ..... CHAN OF storeAnsProtocol ans,
1868 ..... VAL INT ..... type,
1869 ..... VAL [] BYTE ..... data)
1870 .. SEQ
1871 ..... req ! sr.put; type
1872 ..... req ! sr.putdata; (. SIZE data)::data
1873 ..... ans ? sa.ready
1874 ..... put
1875 -----
1876 PROC close(. CHAN OF storeReqProtocol req,
1877 ..... CHAN OF storeAnsProtocol ans)
1878 .. SEQ
1879 ..... req ! srm.close
1880 ..... ans ? sa.ready
1881 ..... close
1882 -----
1883 PROC cancel(. CHAN OF storeReqProtocol req,
1884 ..... CHAN OF storeAnsProtocol ans)
1885 .. SEQ
1886 ..... req ! srm.cancel
1887 ..... ans ? sa.ready
1888 ..... cancel
1889 -----
1890 PROC remove(. CHAN OF storeReqProtocol req,
1891 ..... CHAN OF storeAnsProtocol ans,
1892 ..... VAL INT ..... type,
1893 ..... INT ..... size,
1894 ..... [] BYTE ..... data)
1895 .. SEQ

```

```

1896 ..... get(. req, ans, type, size, data)
1897 ..... CASE type
1898 ..... type.mass
1899 ..... close(. req, ans)
1900 ..... ELSE
1901 ..... SKIP
1902 ..... remove
1903 -----
1904 PROC push(. CHAN OF storeReqProtocol req,
1905 ..... CHAN OF storeAnsProtocol ans,
1906 ..... VAL INT ..... type,
1907 ..... VAL [] BYTE ..... data)
1908 .. SEQ
1909 ..... put(. req, ans, type, data)
1910 ..... CASE type
1911 ..... type.mass
1912 ..... close(. req, ans)
1913 ..... ELSE
1914 ..... SKIP
1915 ..... push
1916 -----
1917 PROC getstat(. CHAN OF storeReqProtocol req,
1918 ..... CHAN OF storeAnsProtocol ans,
1919 ..... INT ..... used, free, all)
1920 .. INT len:
1921 .. [.. 3] INT statvector:
1922 .. SEQ
1923 ..... req ! sr.get; type.stat
1924 ..... [] BYTE statvector RETYPES statvector:
1925 ..... ans ? CASE sa.get; len::statvector
1926 ..... used, free, all := statvector[0], statvector[1], statvector[2]
1927 ..... storestat

```

B.6 Application

app/appifac.inc

```

1928 PROTOCOL APP REQUEST
1929 .. CASE
1930 ..... ar.init; INT::[] BYTE
1931 ..... ar.work; INT::[] BYTE
1932 ..... ar.status; BOOL
1933 ..... ar.close
1934 ..... ar.quit
1935 ..
1936 ..
1937 PROTOCOL APP ANSWER
1938 .. CASE
1939 ..... aa.ready
1940 ..... aa.status
1941 ..... aa.decomp; INT::[] BYTE
1942 ..... aa.results; INT::[] BYTE
1943 ..

```

app/app.occ

```

1944 #INCLUDE "profile/profserv.inc"
1945 #INCLUDE "app/appifac.inc"
1946 #USE "snglmath.lib"
1947 ..
1948 PROC application(. VAL INT ..... NodeId,
1949 ..... CHAN OF APP REQUEST ..... req,
1950 ..... CHAN OF APP ANSWER ..... ans,
1951 ..... CHAN OF profProtocol profChan,
1952 ..... VAL INT ..... maxtaskSize,
1953 ..... [] INT ..... dataarea)
1954 ..
1955 ..
1956 .. PROC rnd(. VAL INT min, max, INT32 seed, INT rndnum) -----
1957 ..... INT32 i:
1958 ..... REAL32 r:

```



```

1959... SEQ
1960... r.i := RAN(.seed)
1961... rndnum := min+(. (. INT i) >> 1) . REM (. (max-min)+1)
1962... rnd
1963-----
1964.. PROC dec(. INT x)
1965... x := x-1
1966..
1967..
1968.. INT nofspawns:
1969.. INT32 seed:
1970..
1971.. PROC appinit()-----
1972... SEQ
1973... INT x:
1974... TIMER t:
1975... SEQ
1976... t ? x
1977... seed := INT32 x
1978... rnd(.10, .20, seed, nofspawns)
1979... evalinit-----
1980..
1981.. PROC appwork()-----
1982... INT spawned:
1983... SEQ
1984... IF
1985... nofspawns > 0
1986... SEQ
1987... rnd(.2, .5, seed, spawned)
1988... SEQ i:=1 FOR spawned
1989... ans ! aa.decomp; 4: "haho"
1990... dec(. nofspawns)
1991... TRUE
1992... SKIP
1993..
1994... TIMER t:
1995... INT a,d:
1996... SEQ
1997... rnd(.100, .500, seed, d)
1998... t ? a
1999... t ? AFTER (. a PLUS d)
2000..
2001... appwork-----
2002..
2003.. BOOL quit:
2004.. SEQ
2005... quit := FALSE
2006... WHILE NOT quit
2007... req ? CASE
2008..
2009... INT init.size:
2010... [] BYTE initdata RETYPES dataarea:
2011... ar.init; init.size::initdata
2012... appinit()
2013..
2014... INT work.size:
2015... [] BYTE workdata RETYPES dataarea:
2016... ar.work; work.size::workdata
2017... SEQ
2018... appwork()
2019... ans ! aa.ready
2020..
2021... ar.close
2022... ans ! aa.results; 1: [0 (. BYTE)]
2023..
2024... ar.quit
2025... quit := TRUE
2026..
2027... application-----

```

B.7 Boss

```
boss/cmmnboss.inc
```

```

2028 PROTOCOL commandProtocol-----
2029.. CASE
2030... stop
2031... tick
2032... cmd; INT; INT... type; direction
2033... commandProtocol-----

```

```
boss/cmmnboss.occ
```

```

2034 #INCLUDE "boss/cmmnboss.inc"
2035 #INCLUDE "store/cmmnstor.inc"
2036 #INCLUDE "profile/profserv.inc"
2037 #INCLUDE "app/appifac.inc"
2038..
2039 PROC cmmnboss(. VAL INT NodeId,
2040... [] CHAN OF commandProtocol com,
2041... [] CHAN OF BOOL ack,
2042... CHAN OF storeReqProtocol req,
2043... CHAN OF storeAnsProtocol ans,
2044... [2] CHAN OF profProtocol profChan,
2045... [] INT memory,
2046... INT dataSize,
2047... VAL INT maxtaskSize)
2048..
2049..
2050... CHAN OF APP.REQUEST app.req:
2051... CHAN OF APP.ANSWER app.ans:
2052..
2053... INT dsize:
2054... SEQ
2055... dsize := dataSize
2056..
2057... INT i.n.t:
2058... [] BYTE i.n.t RETYPES i.n.t:
2059... VAL INT SIZE IS SIZE i.n.t:
2060..
2061... VAL dataSize INT IS (. dsize / INT SIZE)+1:
2062... initdata IS [ memory FROM 0 FOR dataSize INT]:
2063... memory IS [ memory FROM dataSize INT FOR (. SIZE memory)-dataSize INT]:
2064... dataarea IS [ memory FROM 0 FOR (. SIZE memory)-maxtaskSize]:
2065... taskarea IS [ memory FROM (. SIZE dataarea) FOR maxtaskSize]:
2066... PAR
2067... #USE "obj/app.tco"
2068... application(. NodeId, app.req, app.ans, profChan[0], maxtaskSize, dataarea)
2069..
2070... SEQ
2071... #USE "obj/feeder.tco"
2072... feeder(. NodeId, com, ack, req, ans,
2073... profChan[1], app.req, app.ans, initdata, taskarea)
2074..
2075... app.req ! ar.close
2076... [] BYTE results RETYPES initdata:
2077... app.ans ? CASE aa.results; dataSize::results
2078..
2079... app.req ! ar.quit
2080..
2081... cmmnboss-----

```

```
boss/feeder.occ
```

```

2082 #INCLUDE "config/commune.inc"
2083 #INCLUDE "boss/cmmnboss.inc"
2084..
2085 #INCLUDE "store/cmmnstor.inc"
2086 #USE "obj/storifac.tco"
2087..
2088 #INCLUDE "profile/profserv.inc"
2089 #INCLUDE "app/appifac.inc"
2090..

```



```

2091 #USE....."debug.lib"
2092
2093 PROC feeder( VAL INT NodeId,
2094 .....[]CHAN OF commandProtocol com,
2095 .....[]CHAN OF BOOL .....ack,
2096 .....CHAN OF storeReqProtocol req,
2097 .....CHAN OF storeAnsProtocol ans,
2098 .....CHAN OF profProtocol profChan,
2099 .....CHAN OF APP.REQUEST app.req,
2100 .....CHAN OF APP.ANSWER app.ans,
2101 .....[]INT .....initdata,
2102 .....[]INT .....taskarea)
2103
2104 VAL HEAD IS ( NodeId = 0):
2105 VAL NofNeighbours IS ( SIZE com):
2106
2107 []BYTE initdata RETYPES initdata:
2108 []BYTE taskdata RETYPES taskarea:
2109 INT .....tasksize:
2110
2111 BOOL newtask, black, empty, quit:
2112
2113 PROC processtasks()-----
2114 BOOL .....going, spawning:
2115
2116 PROC tick()-----
2117 PAR i=0 FOR MaxNofNeighbours
2118 .....IF
2119 .....i < NofNeighbours
2120 .....SEQ
2121 .....com[i]!.tick
2122 .....TRUE
2123 .....SKIP
2124 .....tick-----
2125
2126 PROC massget()-----
2127 INT available:
2128 SEQ
2129 bookget( req, ans, type.mass, 1, available)
2130 IF
2131 .....available = 1
2132 .....SEQ
2133 .....remove( req, ans, type.mass, tasksize, taskdata)
2134 .....app.req!.ar.work; tasksize:[ taskdata FROM 0 FOR tasksize]
2135 .....spawning, black := TRUE, TRUE
2136 .....TRUE
2137 .....spawning, going, newtask, empty :=
2138 .....FALSE, FALSE, TRUE, TRUE
2139 .....massget-----
2140
2141 PROC massput()-----
2142 INT available:
2143 SEQ
2144 bookput( req, ans, type.mass, 1, available)
2145 IF
2146 .....available = 1
2147 .....SEQ
2148 .....push( req, ans, type.mass, [ taskdata FROM 0 FOR tasksize])
2149 .....spawning, black := TRUE, TRUE
2150 .....TRUE
2151 .....spawning, going, newtask, empty :=
2152 .....FALSE, FALSE, FALSE, FALSE
2153 .....massput-----
2154
2155 SEQ
2156 .....going := TRUE
2157 WHILE going
2158 .....SEQ
2159 .....IF
2160 .....newtask
2161 .....massget()
2162 .....TRUE
2163 .....massput()
2164
2165 WHILE spawning

```

```

2166 .....SEQ
2167 .....tick()
2168 .....app.ans?.CASE
2169 .....aa.ready
2170 .....spawning, going := FALSE, TRUE
2171 .....aa.status
2172 .....INT used, free, all:
2173 .....SEQ
2174 .....getstat( req, ans, used, free, all)
2175 .....app.req!.ar.status; used<3
2176 .....aa.decomp; tasksize:[ taskdata FROM 0 FOR tasksize]
2177 .....massput()
2178
2179 .....processtasks-----
2180
2181 PROC broadcastreq( VAL INT request, type)-----
2182 PAR i=0 FOR MaxNofNeighbours
2183 .....IF
2184 .....i < NofNeighbours
2185 .....BOOL ackflag:
2186 .....SEQ
2187 .....com[i]!.cmd; type; request
2188 .....ack[i]?.ackflag
2189 .....ackflag := NOT ackflag
2190 .....TRUE
2191 .....SKIP
2192 .....broadcastreq-----
2193
2194 PROC puttoken( VAL INT token)-----
2195 BYTE btoken:
2196 SEQ
2197 btoken := BYTE token
2198 put( req, ans, type.token, [ btoken])
2199 .....puttoken-----
2200
2201 PROC gettoken( INT token, VAL INT type)-----
2202 [ 1]BYTE btoken:
2203 INT one:
2204 SEQ
2205 get( req, ans, type.one, btoken)
2206 token := INT btoken[ 0]
2207 .....gettoken-----
2208
2209 INT FUNCTION redtoken( VAL BOOL empty)-----
2210 INT red:
2211 VALOF
2212 .....IF
2213 .....empty
2214 .....red := 1
2215 .....NOT empty
2216 .....red := -1
2217 .....RESULT red
2218 .....redtoken-----
2219
2220 PROC steptoken( INT token)-----
2221 IF
2222 .....HEAD AND ( token < 0)
2223 .....token := token-1
2224 .....HEAD AND ( token > 0)
2225 .....token := token+1
2226 .....TRUE
2227 .....SKIP
2228 .....steptoken-----
2229
2230 PROC sendtoken( VAL INT token)-----
2231 SEQ
2232 .....puttoken( token)
2233 .....com[ HamFwSide]!.cmd; type.token; dir.send
2234 .....BOOL ackflag:
2235 .....ack[ HamFwSide]?.ackflag
2236 .....sendtoken-----
2237
2238 BOOL lastempty:
2239 INT token:
2240

```



```

2241 .. PROC termindetect() -----
2242 .... INT FUNCTION absval(. VAL INT x) -----
2243 .... INT v:
2244 .... VALOF
2245 .... IF
2246 .... x < 0
2247 .... v := -x
2248 .... TRUE
2249 .... v := x
2250 .... RESULT v
2251 .. absval -----
2252 ..
2253 .. SEQ
2254 .. gettoken(. token, . type.token)
2255 .. IF
2256 .. token = 0
2257 .. SKIP
2258 .. TRUE
2259 .. INT used, free, all:
2260 .. SEQ
2261 .. getstat(. req, ans, used, free, all)
2262 .. IF
2263 .. black OR (. empty <> lastempty) OR (. used <> all) AND (. free <> all)
2264 .. token := redtoken(. empty)
2265 .. TRUE
2266 .. SKIP
2267 .. IF
2268 .. HEAD
2269 .. steptoken(. token)
2270 .. TRUE
2271 .. SKIP
2272 ..
2273 .. quit := absval(. token) = 4
2274 .. black, lastempty := FALSE, empty
2275 .. sendtoken(. token)
2276 .. termindetect -----
2277 ..
2278 .. PROC stopguards() -----
2279 .. PAR i=0 FOR MaxNofNeighbours
2280 .. IF
2281 .. i < NofNeighbours
2282 .. SEQ
2283 .. com[i] != stop
2284 .. TRUE
2285 .. SKIP
2286 .. stopguards -----
2287 ..
2288 .. PROC terminfinish() -----
2289 .. SEQ
2290 .. gettoken(. token, . type.wtoken)
2291 .. IF
2292 .. HEAD
2293 .. steptoken(. token)
2294 .. TRUE
2295 .. SKIP
2296 .. sendtoken(. token)
2297 .. IF
2298 .. HEAD
2299 .. SEQ
2300 .. gettoken(. token, . type.wtoken)
2301 .. DEBUG.STOP()
2302 .. TRUE
2303 .. SKIP
2304 ..
2305 .. stopguards()
2306 .. terminfinish -----
2307 ..
2308 .. SEQ
2309 .. lastempty := TRUE
2310 .. IF
2311 .. HEAD
2312 .. puttoken(. redtoken(. lastempty))
2313 .. TRUE
2314 .. SKIP
2315 ..

```

```

2316 .. app.req != ar.init; (. SIZE initdata)::initdata
2317 ..
2318 .. newtask, black, quit := TRUE, FALSE, FALSE
2319 .. WHILE NOT quit
2320 .. SEQ
2321 .. processtasks()
2322 .. INT request:
2323 .. SEQ
2324 .. IF
2325 .. empty
2326 .. request := dir.recv
2327 .. NOT empty
2328 .. request := dir.send
2329 .. broadcastreq(. request, . type.mass)
2330 .. termindetect()
2331 ..
2332 .. terminfinish()
2333 ..
2334 .. feeder -----

```

B.8 Sentinel

sentinel.occ

```

2335 #OPTION "N" --- no usage check
2336 --- buffer is sliced using variable subscripts and assigned to in parallel
2337
2338 #INCLUDE "config/commune.inc"
2339 #INCLUDE "boss/cmmnboss.inc"
2340 #INCLUDE "store/cmmnstor.inc"
2341
2342 -----
2343 PROC sentinel(. VAL INT ..... NodeId,
2344 ..... [CHAN OF INT ..... inp, out,
2345 ..... [CHAN OF commandProtocol ..... com,
2346 ..... [CHAN OF BOOL ..... ack,
2347 ..... [CHAN OF storeReqProtocol ..... sreq,
2348 ..... [CHAN OF storeAnsProtocol ..... sans,
2349 ..... [INT ..... buffer,
2350 ..... VAL INT ..... noifguardbufs, --- = 3
2351 ..... ..... maxtasksize)
2352 .....
2353 .. VAL HEAD IS (. NodeId = 0):
2354 ..
2355 .. VAL NofNeighbours IS (. SIZE inp):
2356 ..
2357 .. #USE "obj/guard.tco"
2358 ..
2359 .. PAR i=0 FOR MaxNofNeighbours
2360 .. IF
2361 .. i < NofNeighbours
2362 .. VAL gbufBegin IS (. i*(maxtasksize * noifguardbufs)):
2363 .. [BYTE gbuffer1 RETYPES [. buffer FROM gbufBegin FOR maxtasksize]:
2364 ..
2365 .. VAL gbufBegin IS gbufBegin+maxtasksize:
2366 .. [BYTE gbuffer2 RETYPES [. buffer FROM gbufBegin FOR maxtasksize]:
2367 ..
2368 .. VAL gbufBegin IS gbufBegin+maxtasksize:
2369 .. [BYTE gbuffer3 RETYPES [. buffer FROM gbufBegin FOR maxtasksize]:
2370 .. guard(. i, inp[i], out[i],
2371 .. com[i], ack[i], sreq[i], sans[i],
2372 .. gbuffer1, gbuffer2, gbuffer3)
2373 .. TRUE
2374 .. SKIP
2375 .. sentinel -----

```

guard.occ

```

2376 #INCLUDE "boss/cmmnboss.inc"
2377
2378 #INCLUDE "store/cmmnstor.inc"

```



```

2379 #USE....."obj/storifac.tco"
2380
2381 #USE....."debug.lib"
2382 #USE....."convert.lib"
2383
2384 -----
2385 PROC guard( VAL INT..... guardid,
2386 ..... CHAN OF INT..... inp, out,
2387 ..... CHAN OF commandProtocol com,
2388 ..... CHAN OF BOOL..... ack,
2389 ..... CHAN OF storeReqProtocol req,
2390 ..... CHAN OF storeAnsProtocol ans,
2391 ..... [] BYTE..... inp.buf1, inp.buf2, out.buf)
2392
2393 .. PROTOCOL sentinelProtocol -----
2394 ... CASE
2395 ..... init; INT; INT..... init; REQTYPE; REQUEST
2396 ..... xfer; INT:: [] BYTE
2397 ..:.. sentinelProtocol -----
2398
2399 .. CHAN OF sentinelProtocol inp RETYPES inp:
2400 .. CHAN OF sentinelProtocol out RETYPES out:
2401
2402 .. BOOL quit:
2403 .. INT typeInp, typeOut, dirInp, dirOut:
2404
2405 .. BOOL tie:
2406 .. INT massDirOut:
2407
2408 .. INT FUNCTION inverseDir( VAL INT dir) -----
2409 ... INT idir:
2410 ... VAL OF
2411 ..... CASE dir
2412 ..... dir.send
2413 ..... idir := dir.recv
2414 ..... dir.recv
2415 ..... idir := dir.send
2416 ..... RESULT idir
2417 ..... inverseDir -----
2418
2419 .. PROC transfer( VAL INT dir.1, dir.2, type.1, type.2) -----
2420 ... INT inp.size1, inp.size2, out.size:
2421 ... INT dir1, dir2, type1, type2:
2422 ... SEQ
2423 ... IF
2424 ..... type.1 <= type.2
2425 ..... type1, type2, dir1, dir2 := type.1, type.2, dir.1, dir.2
2426 ..... TRUE
2427 ..... type1, type2, dir1, dir2 := type.2, type.1, dir.2, dir.1
2428
2429 ..... VAL flagInp1 IS ( dir1=dir.recv):
2430 ..... VAL flagInp2 IS ( dir2=dir.recv) AND ( ( type1<type2) OR ( dir1<dir2)):
2431 ..... VAL flagOut1 IS ( dir1=dir.send):
2432 ..... VAL flagOut2 IS ( dir2=dir.send) AND ( ( type1<type2) OR ( dir1<dir2)):
2433 ..... SEQ
2434 ..... PAR
2435 ..... SEQ --- inp -----
2436 ..... IF
2437 ..... flagInp1
2438 ..... inp ? CASE xfer; inp.size1::inp.buf1
2439 ..... TRUE
2440 ..... SKIP
2441 ..... IF
2442 ..... flagInp2
2443 ..... inp ? CASE xfer; inp.size2::inp.buf2
2444 ..... TRUE
2445 ..... SKIP
2446 ..... SEQ --- out -----
2447 ..... IF
2448 ..... flagOut1
2449 ..... SEQ
2450 ..... remove( req, ans, type1, out.size, out.buf)
2451 ..... out ! xfer; out.size::out.buf
2452 ..... TRUE
2453 ..... SKIP

```

```

2454 ..... IF
2455 ..... flagOut2
2456 ..... SEQ
2457 ..... remove( req, ans, type2, out.size, out.buf)
2458 ..... out ! xfer; out.size::out.buf
2459 ..... TRUE
2460 ..... SKIP
2461
2462 ..... storing incoming data -----
2463 ..... IF
2464 ..... flagInp1
2465 ..... push( req, ans, type1, [ inp.buf1 FROM 0 FOR inp.size1])
2466 ..... TRUE
2467 ..... SKIP
2468 ..... IF
2469 ..... flagInp2
2470 ..... push( req, ans, type2, [ inp.buf2 FROM 0 FOR inp.size2])
2471 ..... TRUE
2472 ..... SKIP
2473 ..:.. transfer -----
2474
2475 .. INT booked:
2476 .. BOOL xferflag:
2477
2478 .. PROC handshake() -----
2479 ... PROC corrector() -----
2480 ... IF
2481 ..... is.mass( typeOut) AND is.mass( typeInp) AND ( dirOut = dirInp)
2482 ..... SEQ
2483 ..... cancel( req, ans)
2484 ..... xferflag, tie, massDirOut := FALSE, TRUE, dirOut
2485
2486 ..... is.mass( typeInp) AND is.ctrl( typeOut)
2487 ..... INT success:
2488 ..... SEQ
2489 ..... book( req, ans, type.mass, inverseDir( dirInp), 1, success)
2490 ..... IF
2491 ..... success = 1
2492 ..... SEQ
2493 ..... out ! init; type.mass; inverseDir( dirInp)
2494 ..... tie := FALSE
2495 ..... TRUE
2496 ..... SEQ
2497 ..... out ! init; type.mass; dirInp
2498 ..... tie, massDirOut := TRUE, dirInp
2499 ..... typeInp, dirInp := typeOut, inverseDir( dirOut)
2500
2501 ..... is.ctrl( typeInp) AND is.mass( typeOut)
2502 ..... INT type, dir:
2503 ..... SEQ
2504 ..... inp ? CASE init; type; dir
2505 ..... IF
2506 ..... dir = dirOut
2507 ..... SEQ
2508 ..... cancel( req, ans)
2509 ..... tie, massDirOut := TRUE, dirOut
2510 ..... typeOut, dirOut := typeInp, inverseDir( dirInp)
2511 ..... TRUE
2512 ..... SKIP
2513 ..... TRUE
2514 ..... SKIP
2515 ..:.. corrector -----
2516
2517 ... SEQ
2518 ..... book( req, ans, typeOut, dirOut, 1, booked)
2519 ..... xferflag := ( booked = 1)
2520 ..... IF
2521 ..... booked = 1
2522 ..... SEQ
2523 ..... PAR
2524 ..... out ! init; typeOut; dirOut
2525 ..... inp ? CASE init; typeInp; dirInp
2526 ..... corrector()
2527 ..... IF
2528 ..... xferflag

```



```

2529 ..... transfer(. inverseDir(. dirInp), . dirOut, . typeInp, . typeOut)
2530 ..... TRUE
2531 ..... SKIP
2532 ..... TRUE
2533 ..... SKIP
2534 :--- handshake -----
2535
2536 .. SEQ
2537 ... quit, . tie, . massDirOut := FALSE, . TRUE, . dir.recv
2538
2539 ... WHILE NOT quit
2540 ..... SEQ
2541 ..... PRI ALT
2542 ..... com ? CASE
2543 ..... stop
2544 ..... quit := TRUE
2545 ..... tick
2546 ..... IF
2547 ..... tie
2548 ..... SEQ
2549 ..... typeOut, . dirOut := type.mass, . inverseDir(. massDirOut)
2550 ..... handshake()
2551 ..... TRUE
2552 ..... SKIP
2553 ..... cmd: typeOut; . dirOut
2554 ..... SEQ
2555 ..... handshake()
2556 ..... ack ! tie
2557
2558 ..... inp ? CASE
2559 ..... init; . typeInp; . dirInp
2560 ..... SEQ
2561 ..... book(. req, . ans, . typeInp, . inverseDir(. dirInp), . 1, . booked)
2562 ..... IF
2563 ..... booked =- 1
2564 ..... SEQ
2565 ..... dirOut, . typeOut := inverseDir(. dirInp), . typeInp
2566 ..... out ! init; . typeOut; . dirOut
2567 ..... transfer(. dirOut, . dirOut, . typeOut, . typeOut)
2568 ..... TRUE
2569 ..... SEQ
2570 ..... dirOut, . typeOut := dirInp, . typeInp
2571 ..... out ! init; . typeOut; . dirOut
2572 ..... tie := TRUE
2573
2574 ..... --- for debugging
2575 ..... INT size:
2576 ..... [10000] BYTE buf:
2577 ..... xfer; size::buf
2578 ..... DEBUG.STOP()
2579
2580 :--- guard -----

```


Appendix C

NOP executor code listing

Makefile

```
1 SHELL=/bin/sh
2 FLEX=flex
3 YACC=bison
4 CC=cc
5 OBJDIR=obj
6
7 LDFLAGS=-g
8 CFLAGS=-Wall -g
9
10 .SUFFIXES:
11
12 objects := $(patsubst %.c,$(OBJDIR)/%.o,$(wildcard *.c))
13 objects := $(objects) $(OBJDIR)/nop.tab.o $(OBJDIR)/noplex.o
14
15 $(OBJDIR)/%.o: %.c
16     $(CC) -c $(CFLAGS) $< -o $@
17
18 all: nopce
19
20 nopce: $(objects)
21     $(CC) $(LDFLAGS) $~ -o $@
22
23 $(OBJDIR)/nop.tab.h $(OBJDIR)/nop.tab.c: nop.y
24     $(YACC) -tv -b $(OBJDIR)/nop $<
25
26 $(OBJDIR)/nop.tab.o: $(OBJDIR)/nop.tab.c
27     $(CC) -c $(CFLAGS) $< -o $@
28
29 $(OBJDIR)/noplex.c: nop.lex
30     $(FLEX) -o$@ $<
31
32 $(OBJDIR)/noplex.o: $(OBJDIR)/noplex.c
33     $(CC) -c $(CFLAGS) $< -o $@
34
35 clean:
36     rm -f $(OBJDIR)/*
37     rm -f nopce.*
38     rm -f *.bak
39     rm -f *.rep
40
41 build:
42     make clean
43     make all
44
45
```

```
46 MKDEPFLAGS=-f Depfile -Y -p$(OBJDIR)/
47
48 depend:
49     makedepend $(MKDEPFLAGS) *.c >&/dev/null
50     makedepend -a $(MKDEPFLAGS) -o.tab.c *.y >&/dev/null
51     makedepend -a $(MKDEPFLAGS) -o.lex.c *.lex >&/dev/null
52
53 #include Depfile
```

nop.h

```
54 #ifndef nop_included
55 #define nop_included
56
57 typedef unsigned char Byte;
58 typedef unsigned int UInt;
59 typedef enum { FALSE=0, NO=0, TRUE=1, YES=1 } Bool;
60
61 #define NOT !
62 #define AND &&
63 #define OR ||
64
65 #endif // nop_included
```

nop.c

```
66 #include <stdio.h>
67 #include <string.h>
68 #include "nop.h"
69
70 #include "memnode.h"
71 #include "memexec.h"
72 #include "memstack.h"
73
74 #include "atomtab.h"
75 #include "trafotab.h"
76
77 #include "reprpars.h"
78 #include "nopexec.h"
79 #include "noptrav.h"
80
81 FILE *report;
82 char repstr[4096];
83
84 void printreport( char *s) {
85     fputs( s, report);
```



```

86 }
87
88 void printerror( char *s) {
89     printreport( s);
90     fputs( s, stderr);
91 }
92
93 void printabort( char *s) {
94     printerror( s);
95     abort();
96 }
97
98 int main( int argc, char **argv) {
99
100     ++argv, --argc;
101     if ( argc > 0 )
102         report = fopen( argv[ 0], "w");
103     else
104         report = fopen( "nop.rep", "w");
105
106     if ( report == NULL) {
107         fputs( "unable to open report file\n", stderr);
108         abort();
109     }
110
111     yyin = fopen( "nopce.cpp", "r");
112     if ( yyin == NULL)
113         printabort( "unable to open input file 'nopce.cpp'\n");
114
115     printerror( "parsing...\n");
116     if ( parser()) {
117         printerror( "parse failed\n");
118         exit( EXIT_FAILURE);
119     }
120
121     fclose( yyin);
122
123     printreport( "\natom table:\n");
124     atomtable_list();
125     printreport( "\ntrafo name table:\n");
126     trafotable_list();
127     printreport( "\ndefinitions: \n");
128     printdefs();
129     printreport( "=====\n");
130
131     printerror( "checking definitions...\n");
132     checkdefs();
133     printreport( "\ndefinitions processed: \n");
134     printdefs();
135     printreport( "=====\n");
136
137     printerror( "translation...\n");
138     translate();
139     printreport( "\nprogram translated: \n");
140     printprog();
141     printreport( "=====\n");
142
143     fclose( report);
144
145     execute();
146
147     exit( EXIT_SUCCESS);
148 }

```

C.1 Data representation

```
reprnode.h
```

```

149 #ifndef reprnode_included
150 #define reprnode_included
151

```

```

152 #include "nop.h"
153
154 typedef Uint          Atom;
155
156 typedef struct Node_tag Node;
157 typedef Node*         PNode;
158
159 typedef union Value_tag {
160     Atom  atom;
161     PNode pnode;
162 } Value;
163
164 typedef Value*         PValue;
165
166 struct Node_tag {
167     Value  value;
168     PNode  pfirstsub;
169     PNode  pprev, pnext;
170 }; // Node
171
172 Bool  isatom( Value value);
173 Bool  isnode( Value value);
174 Value setatom( Atom thisatom);
175 Atom  getatom( Value value);
176 Value setnode( PNode pthisnode);
177 PNode getnode( Value value);
178
179 #endif

```

```
reprnode.c
```

```

180 #include "reprnode.h"
181 #include "nopalloc.h"
182 #include "memnode.h"
183
184 #define BECAREFUL
185
186 Bool isatom( Value value) {
187     return value.atom & 1;
188 }
189
190 Bool isnode( Value value) {
191     return ! isatom( value);
192 }
193
194 Value setatom( Atom thisatom) {
195     return ( Value) ( ( thisatom << 1) | 1);
196 }
197
198 Atom getatom( Value value) {
199     return ( Atom) ( ( value.atom >> 1);
200 }
201
202 Value setnode( PNode pthisnode) {
203
204     #ifndef BECAREFUL
205     #include <stdio.h>
206     if ( ( ( Uint) pthisnode) & 1) {
207         fprintf( stderr, "oops, odd pointer: %p\n", pthisnode);
208         exit( 1);
209     }
210     else
211     #endif
212         return ( Value) pthisnode;
213 }
214
215 PNode getnode( Value value) {
216     return value.pnode;
217 }

```


C.2. TRANSFORMATION REPRESENTATION

C.2 Transformation representation

reprtraf.h

```

218 #ifndef reprtraf_included
219 #define reprtraf_included
220
221 #include "nop.h"
222 #include "reprnode.h"
223 #include "strtable.h"
224
225 typedef StrTableCode      Name;
226
227 typedef enum {
228     BC_exch = 0,
229     BC_cdn,
230     BC_cup,
231     BC_insert,
232     BC_delete,
233     BC_left,
234     BC_right,
235     BC_split,
236     BC_replace,
237     BC_skip,
238     BC_seq,
239     BC_equ,
240     BC_all,
241 // extra
242     BC_trace,
243     BC_trace0,
244     BC_trace1,
245     BC_dump
246 } BinCode;
247
248 #define NOF_BUILTINS      ( BC_dump+1)
249
250 typedef struct {
251     Name namecode;
252     Uint arity;
253 } Builtins;
254
255 extern Builtins builtin[ NOF_BUILTINS];
256
257 typedef struct Trafo*      PTrafo;
258
259 typedef enum {
260     TT_SET,
261     TT_BIN,
262     TT_NAME,
263     TT_LOOP
264 } TrafoType;
265
266 typedef struct Trafo {
267     Bool      ready;
268     Uint      refid;
269     TrafoType type;
270     union {
271         Value      value;          // TT_SET
272         struct {
273             BinCode bc;
274             PTrafo* args;
275         }          // TT_BIN
276         struct {
277             Name name;
278             Uint nofargs;
279             PTrafo* args;
280         }          // TT_NAME
281         PTrafo loop;          // TT_LOOP
282     }
283 } Trafo;
284
285 extern PTrafo      program;
286
287 void checkdefs( void);

```

```

288 void translate( void);
289
290 #endif

```

reprtraf.c

```

291 #include <stdio.h>
292 #include "nop.h"
293 #include "nopalloc.h"
294 #include "reprtraf.h"
295 #include "reprpars.h"
296 #include "traftab.h"
297 #include "noptrav.h"
298
299 typedef struct THistory*      PTHistory;
300 typedef struct THistory {
301     Name      name;
302     PTrafo    ptrafa;
303     PTHistory prev;
304 } THistory;
305
306 static PTHistory      phistory;
307 static Uint           refcnt;
308
309 static void hpush( Name name, Uint arity, PTrafo ptrafa, Uint *pnum) {
310     PTHistory      ph;
311     if ( arity > 0)
312         return;
313
314     ++*pnum;
315     ph = phistory;
316     phistory = nopalloc( sizeof *phistory);
317     phistory->name = name;
318     phistory->ptrafa = ptrafa;
319     phistory->prev = ph;
320 }
321
322 static void hpop( void) {
323     PTrafo      ptrafa;
324     PTHistory    ph;
325     if ( phistory == NULL)
326         abort();
327     ptrafa = phistory->ptrafa;
328     ph = phistory;
329     phistory = phistory->prev;
330     free( ph);
331 }
332
333 static PTrafo hcheck( Name name, Uint arity) {
334     PTHistory    ph;
335     if ( arity > 0)
336         return NULL;
337     for ( ph = phistory; ph != NULL; ph = ph->prev)
338         if ( ph->name == name)
339             return ph->ptrafa;
340     return NULL;
341 }
342
343 static PTrafo newtrafo( TrafoType type) {
344     PTrafo      ptrafa;
345     ptrafa = nopalloc( sizeof *ptrafa);
346     ptrafa->ready = NO;
347     ptrafa->refid = 0;
348     ptrafa->type = type;
349     return ptrafa;
350 }
351
352 static void refize( PTrafo ptrafa) {
353     if ( ptrafa->refid == 0)

```



```

360 ptrafa->refid = ++refcnt;
361 }
362
363 static void subst( PTDef ptdef, PTrafo ptrafa) {
364     Bool loop;
365
366     PTrafo resolve( Name name) {
367         Uint i;
368
369         for ( i = 0; i < ptdef->header.params.nofparams; i++)
370             if ( name == ptdef->header.params.paramv[ i].name)
371                 return ptrafa->select.name.args[ i];
372         return NULL;
373     }
374
375     PTrafo xxcopy( PTRef ptref) {
376         PTrafo pt;
377
378         switch ( ptref->reftype) {
379             case REFT_SET:
380                 pt = newtrafo( TT_SET);
381                 pt->select.value = ptref->select.value;
382                 break;
383             case REFT_NAME: {
384                 Uint i, arity;
385                 int lidx;
386                 Name name;
387
388                 name = ptref->select.refname.name;
389                 lidx = ptref->select.refname.localidx;
390                 arity = ptref->select.refname.args.nofargs;
391
392                 if ( lidx >= 0) {
393                     pt = ptrafa->select.name.args[ lidx];
394                     if ( arity == 0) {
395                         if ( ptdef->header.params.paramv[ lidx].refcnt > 1)
396                             refize( pt);
397                         break;
398                     }
399                     else {
400                         if ( ( pt->type != TT_NAME) ||
401                             ( pt->select.name.nofargs != 0)) {
402                             printerror( "invalid construction\n");
403                             printtrafo( "subtrafo -----< n");
404                             printtrafo( pt, printerror);
405                             printerror( "reference -----< n");
406                             printref( ptref, 0, printerror);
407                             printerror( "n-- def -----< n");
408                             printdef( ptdef, printerror);
409                             printerror( "n-- trafo -----< n");
410                             printtrafo( ptrafa, printerror);
411                             abort();
412                         }
413                         else
414                             name = pt->select.name.name;
415                     }
416                 }
417                 pt = newtrafo( TT_NAME);
418                 pt->select.name.name = name;
419                 pt->select.name.nofargs = arity;
420                 pt->select.name.args = nopalloc( arity * sizeof ( PTrafo));
421                 for ( i = 0; i < arity; i++)
422                     pt->select.name.args[ i] =
423                         xxcopy( ptref->select.refname.args.argv[ i]);
424                 break;
425             }
426             case REFT_LOOP:
427                 pt = newtrafo( TT_LOOP);
428                 pt->select.loop = ptrafa;
429                 loop = YES;
430                 break;
431         }
432         return pt;
433     }
434     PTrafo ptrafa;

```

```

435     loop = NO;
436     pt = xxcopy( ptdef->select.tref);
437     pt->refid = ptrafa->refid;
438     *ptrafa = *pt;
439     if ( loop)
440         refize( ptrafa);
441     free( pt);
442 }
443
444 static PTDef deflookup( Name name, Uint arity) {
445     Uint i;
446     PTDef p;
447
448     for ( i = 0; i < definitions.nofdefs; i++) {
449         p = definitions.defv[ i];
450         if ( ( p->header.name == name) &&
451             ( p->header.params.nofparams == arity))
452             return p;
453     }
454     return NULL;
455 }
456
457 static void substitute( PTrafo ptrafa) {
458     PTDef ptdef;
459     Name name;
460     Uint arity;
461
462     name = ptrafa->select.name.name;
463     arity = ptrafa->select.name.nofargs;
464
465     ptdef = deflookup( name, arity);
466     if ( ptdef == NULL) {
467         sprintf( repstr, "undefined reference: %s/%d\n",
468             trafoable_search( name), arity);
469         printerror( repstr);
470         printerror( "n-- trafo -----< n");
471         printtrafo( ptrafa, printerror);
472         abort();
473     }
474     else {
475         switch ( ptdef->deftype) {
476             case DEFT_BUILTIN:
477                 ptrafa->type = TT_BIN;
478                 ptrafa->select.bin.args = ptrafa->select.name.args;
479                 ptrafa->select.bin.bc = ptdef->select.bincode;
480                 break;
481             case DEFT_USER:
482                 subst( ptdef, ptrafa);
483                 break;
484         }
485     }
486 }
487
488 static void expand( PTrafo ptrafa) {
489     Uint popnum;
490
491     popnum = 0;
492
493     while ( NOT ptrafa->ready) {
494         //puts( "n-- current trafo -");
495         //printtrafo( ptrafa);
496
497         switch ( ptrafa->type) {
498             case TT_SET:
499                 ptrafa->ready = YES;
500                 break;
501             case TT_BIN: {
502                 Uint i, arity;
503
504                 arity = builtin[ ptrafa->select.bin.bc].arity;
505                 for ( i = 0; i < arity; i++)
506                     expand( ptrafa->select.bin.args[ i]);
507             }
508         }
509     }

```



```

510     ptrafa->ready = YES;
511     break;
512 }
513 case TT_NAME: {
514     Name     name;
515     Uint     arity;
516     PTrafo   pt;
517
518     name = ptrafa->select.name.name;
519     arity = ptrafa->select.name.nofargs;
520
521     pt = hcheck( name, arity);
522     if ( pt != NULL) {
523         ptrafa->type = TT_LOOP;
524         ptrafa->select.loop = pt;
525         refize( pt);
526     }
527     else {
528         hpush( name, arity, ptrafa, &popnum);
529         substitute( ptrafa);
530     }
531     break;
532 }
533 case TT_LOOP:
534     ptrafa->ready = YES;
535     break;
536 }
537 }
538
539 while ( popnum > 0) {
540     hpop();
541     popnum--;
542 }
543 }
544
545 PTrafo program;
546
547 void checkdefs( void) {
548     PDef ptdef;
549
550     void traverse( PRef ptref) {
551         Uint i;
552
553         switch ( ptref->reftype) {
554             case REFT_SET:
555                 break;
556             case REFT_NAME:
557                 ptref->select.refname.localidx = -1;
558                 for ( i = 0; i < ptdef->header.params.nofparams; i++)
559                     if ( ptref->select.refname.name ==
560                         ptdef->header.params.paramv[ i].name) {
561                         ptref->select.refname.localidx = i;
562                         ++ptdef->header.params.paramv[ i].refcnt;
563                         break;
564                     }
565
566                 for ( i = 0; i < ptref->select.refname.args.nofargs; i++)
567                     traverse( ptref->select.refname.args.argv[ i]);
568
569                 break;
570             case REFT_LOOP:
571                 break;
572         }
573     }
574
575     Uint i, j;
576
577     for ( i = 0; i < definitions.nofdefs; i++) {
578         ptdef = definitions.defv[ i];
579
580         switch ( ptdef->deftype) {
581             case DEFT_BUILTIN:
582                 break;
583             case DEFT_USER:
584                 traverse( ptdef->select.tref);

```

```

585         for ( j = 0; j < ptdef->header.params.nofparams; j++)
586             if ( ptdef->header.params.paramv[ j].refcnt == 0) {
587                 sprintf( repstr, "%s/%d: parameter %s not referenced\n",
588                     trafotable_search( ptdef->header.name),
589                     ptdef->header.params.nofparams,
590                     trafotable_search( ptdef->header.params.paramv[ j].name));
591                 perror( repstr);
592             }
593         break;
594     }
595 }
596
597 void translate( void) {
598     phistory = NULL;
599     refcnt = 0;
600
601     program = newtrafo( TT_NAME);
602     program->select.name.name = trafotable_lookup( "main");
603     program->select.name.nofargs = 0;
604     program->select.name.args = NULL;
605     expand( program);
606 }
607
608 }

```

C.3 Memory management

nopalloc.h

```

609 #ifndef nopalloc_included
610 #define nopalloc_included
611
612 #include <stdlib.h>
613
614 void *nopalloc( size_t size);
615 void *noprealloc( void *p, size_t newsize);
616
617 #endif

```

nopalloc.c

```

618 #include <stdio.h>
619 #include "nopalloc.h"
620
621 void *nopalloc( size_t size) {
622     register void *p;
623
624     if ( ( p = malloc( size)) == NULL) {
625         fprintf( stderr, "nopalloc\n");
626         abort();
627     }
628     return p;
629 }
630
631 void *noprealloc( void *p, size_t newsize) {
632     if ( ( p = realloc( p, newsize)) == NULL) {
633         fprintf( stderr, "noprealloc\n");
634         abort();
635     }
636     return p;
637 }

```

mem.h

```

638 #ifndef mem_included
639 #define mem_included
640
641 #include "nop.h"
642 #include <stdlib.h>
643

```



```

644 typedef void *Pvoid;
645 typedef Pvoid *PPtr;
646
647 typedef void ( *CBMemFunc)( Pvoid);
648
649 Pvoid nopmeminit( size_t total, size_t size);
650 Pvoid nopmemget( PPtr pbase);
651 void nopmemput( PPtr pbase, Pvoid pitem);
652 Bool nopmemmove( PPtr dest, PPtr src);
653 Bool nopmemfmove( PPtr dest, PPtr src, CBMemFunc cbmf);
654
655 #endif

```

mem.c

```

656 #include "mem.h"
657
658 Pvoid nopmeminit( size_t total, size_t itemsize) {
659     Pvoid p;
660     PPtr q, r;
661     size_t i, size;
662
663     if ( ( size = total * itemsize) > 0) {
664         p = malloc( size);
665         if ( p != NULL) {
666             for ( i = 0, q = p; i < total-1; i++, q = r)
667                 *q = r = ( ( Pvoid) q)+itemsize;
668             *q = NULL;
669         }
670         return p;
671     }
672     else
673         return NULL;
674 } // nopmeminit
675
676 // removes an item from the head of a list
677 // pbase != NULL
678 Pvoid nopmemget( PPtr pbase) {
679     register PPtr p;
680
681     if ( ( p = *pbase) != NULL) {
682         *pbase = *p;
683         *p = NULL; // should not be necessary
684     }
685     return p;
686 }
687
688 // attaches an item to the front of a list
689 // pbase, pitem != NULL
690 void nopmemput( PPtr pbase, Pvoid pitem) {
691     *( PPtr) pitem = *pbase;
692     *pbase = pitem;
693 }
694
695 // moves an item head of list src to front of list dest
696 Bool nopmemmove( PPtr pdest, PPtr psrc) {
697     register PPtr p;
698
699     if ( ( p = nopmemget( psrc)) != NULL)
700         nopmemput( pdest, p);
701
702     return p != NULL;
703 }
704
705 Bool nopmemfmove( PPtr pdest, PPtr psrc, CBMemFunc cbmf) {
706     Bool success;
707
708     if ( ( success = nopmemmove( pdest, psrc)))
709         (*cbmf)( *pdest);
710
711     return success;
712 }

```

memexec.h

```

713 #include "mem.h"
714 #include "reprtraf.h"
715 #include "reprnode.h"
716 #include "memstack.h"
717
718 typedef struct ExecUnit_tag *PExecUnit;
719 typedef struct ExecUnit_tag {
720     Pvoid chainptr; // needed for mem... to work on active list
721     PNode pnode;
722     PStackItem pstack;
723     PExecUnit parent;
724     Uint childcnt;
725 } ExecUnit;
726
727 Bool execuinit( size_t total);
728 Bool execucrate( PExecUnit template);
729 PExecUnit execuget( void);
730 void execufree( PExecUnit peunit);
731 void execurevive( PExecUnit peunit);

```

memexec.c

```

732 #include "memexec.h"
733
734 static PExecUnit eunitactivelist, eunitfreelist;
735
736 Bool execuinit( size_t total) {
737     eunitactivelist = NULL;
738     eunitfreelist = ( PExecUnit) nopmeminit( total, sizeof ( ExecUnit));
739     return eunitfreelist != NULL;
740 }
741
742 // free -> active
743 Bool execucrate( PExecUnit template) {
744     if ( nopmemmove( ( PPtr) &eunitactivelist, ( PPtr) &eunitfreelist)) {
745         // eunitactivelist->chainptr = DO NOT MODIFY !!!!;
746         eunitactivelist->pnode = template->pnode;
747         eunitactivelist->pstack = template->pstack;
748         eunitactivelist->parent = template->parent;
749         eunitactivelist->childcnt = template->childcnt;
750         return YES;
751     }
752     return NO;
753 }
754
755 // active -> used
756 PExecUnit execuget( void) {
757     return nopmemget( ( PPtr) &eunitactivelist);
758 }
759
760 // used -> free
761 void execufree( PExecUnit peunit) {
762     nopmemput( ( PPtr) &eunitfreelist, peunit);
763 }
764
765 // used -> active
766 void execurevive( PExecUnit peunit) {
767     nopmemput( ( PPtr) &eunitactivelist, peunit);
768 }

```

memnode.h

```

769 #include "mem.h"
770 #include "reprnode.h"
771
772 Bool nodememinit( size_t total);
773 PNode nodealloc( void);
774 PNode nodecreate( Value value, PNode pndelist);
775 void nodefill( PNode pnode, Value value, PNode pndelist);
776 PNode nodeallocfill( Value value, PNode pndelist);
777 PNode nodelistappend( PNode pndelist, PNode pnode);
778 void nodefree( PNode pnode);

```

memnode.c


```

779 #include "nopalloc.h"
780 #include "memnode.h"
781
782 static PNode nodefreelist;
783
784 Bool nodememinit( size_t total) {
785     nodefreelist = ( PNode) nopmeminit( total, sizeof ( Node));
786     return nodefreelist != NULL;
787 }
788
789 PNode nodealloc( void) {
790     return nopmemget( ( PPtr) &nodefreelist);
791 }
792
793 void nodefill( PNode pnode, Value value, PNode pndolist) {
794     pnode->pprev = pnode;
795     pnode->pnext = pnode;
796     pnode->value = value;
797     pnode->pfirstsub = pndolist;
798 }
799
800 PNode nodecreate( Value value, PNode pndolist) {
801     PNode pn;
802
803     pn = nopalloc( sizeof *pn);
804     nodefill( pn, value, pndolist);
805     return pn;
806 }
807
808 PNode nodeallocfill( Value value, PNode pndolist) {
809     PNode pnode;
810
811     pnode = nodealloc();
812     nodefill( pnode, value, pndolist);
813     return pnode;
814 }
815
816 PNode nodelistappend( PNode pndolist, PNode pnode) {
817     if ( pndolist == NULL)
818         return pnode;
819     else {
820         pndolist->pprev->pnext = pnode;
821         pnode->pprev = pndolist->pprev;
822         pndolist->pprev = pnode;
823         pnode->pnext = pndolist;
824     }
825     return pndolist;
826 }
827
828 void nodefree( PNode pnode) {
829     nopmemp( ( PPtr) &nodefreelist, pnode);
830 }

```

memstack.h

```

831 #ifndef memstack_included
832 #define memstack_included
833
834 #include "mem.h"
835 #include "reprtraf.h"
836
837 typedef struct StackItem_tag *PStackItem;
838 typedef struct StackItem_tag {
839     PStackItem prev; // must be the first member
840     PTrafo ptraf;
841 } StackItem;
842
843 Bool stackinit( size_t total);
844 Bool stackpush( PStackItem *psp, PTrafo ptraf);
845 PTrafo stackpop( PStackItem *psp);
846
847 #endif

```

memstack.c

```

848 #include "memstack.h"
849
850 static PStackItem stackfreelist;
851
852 Bool stackinit( size_t total) {
853     stackfreelist = ( PStackItem) nopmeminit( total, sizeof ( StackItem));
854     return stackfreelist != NULL;
855 }
856
857 Bool stackpush( PStackItem *psp, PTrafo ptraf) {
858     if ( nopmemmove( ( PPtr) psp, ( PPtr) &stackfreelist)) {
859         (*psp)->ptraf = ptraf;
860         return YES;
861     }
862     return NO;
863 }
864
865 PTrafo stackpop( PStackItem *psp) {
866     if ( nopmemmove( ( PPtr) &stackfreelist, ( PPtr) psp))
867         return stackfreelist->ptraf;
868     return NULL;
869 }

```

C.4 Translation

nop.lex

```

870 /* lexical analyser */
871
872 %option noyywrap
873 /* %option stack */
874
875 %{
876 #include <stdlib.h>
877 #include <ctype.h>
878 #include "lexparse.h"
879 #include "obj/nop.tab.h"
880
881 #define ESCAPE '#'
882
883 #define YY_USER_ACTION lex_linepos += yyleng;
884 //printf( "\nlex:>%s< l:%ld, p:%ld\n", yytext, lex_linenum, lex_linepos);
885
886 void checkvaluemod( void);
887
888 void setfilename( void) {
889     Uint i;
890
891     free( lex_filename);
892     lex_filename = nopalloc( yyleng);
893     for ( i = 0; i < yyleng-2; i++)
894         lex_filename[i] = yytext[i+1];
895     lex_filename[i] = '\0';
896
897     //printf( "\n>>>>>>%s<<<<<<<<<<\n", lex_filename);
898 }
899
900 Uint unesc( char* src) {
901     Uint len;
902     char *dest;
903
904     dest = src;
905     for ( len = 0; *src != '\0'; len++, src++)
906         if ( *src != ESCAPE)
907             *dest++ = *src;
908         else
909             *dest++ = **src;
910     *dest = '\0';
911
912     return len;
913 }

```



```

914 void atomcode( void) {
915     Uint xval( char c) {
916         Uint x = 0;
917
918         switch ( c) {
919             case '0'...'9': x = c-'0'; break;
920             case 'A'...'F': x = c-'A'+10; break;
921             case 'a'...'f': x = c-'a'+10; break;
922         }
923         return x;
924     }
925
926     yylval.sv_name = 16*xval( yytext[ 1]) + xval( yytext[ 2]);
927     //printf( "****atom: %d\n", yylval.sv_name);
928 }
929
930 void atom( void) {
931     char* s;
932
933     s = strcpy( nopalloc( unesc( yytext)+1), yytext);
934     yylval.sv_name = atombtable_insert( s);
935     //printf( "****atom: %d\n", yylval.sv_name);
936 }
937
938 void trafo( void) {
939     char* s;
940
941     s = strcpy( nopalloc( yleng+1), yytext);
942     yylval.sv_name = trafotable_insert( s);
943 }
944
945 %%
946
947 LINENUM    [[:digit:]]+
948 FILENAME   \"[~\\\"\\n]+\"
949
950
951 TRAFOCHAR  [[:alnum:]]|[_]
952 TRAFOID    [[:alpha:]]{TRAFOCHAR}*
953
954 ATOMNORM   [~\\0-\\x20#'\\\"()]
955 ATOMESC    #[#'\\\"()]
956
957 SATOMNORM  [~\\0-\\x1f#\\\"]
958 SATOMESC   #[#\\\"]
959
960 ATOMCODE   #[[[:xdigit:]]|[:xdigit:]]
961
962 ATOMC      {ATOMNORM}|{ATOMESC}
963 SATOMC     {SATOMNORM}|{SATOMESC}
964
965 ATOMSTR    {ATOMC}+
966
967 ANY        (.|\\n)
968 BLANK      [\\t\\f ]+
969
970 %x VALUE VALUE_STRING
971 %x PREPROC
972
973 %%
974
975     checkvaluemod();
976     lex_linenum++; lex_linepos = 0;
977 {BLANK}    /* empty */
978
979 ""# "{LINENUM} lex_linenum = atoi( yytext+2); BEGIN( PREPROC);
980
981 <PREPROC>{
982     {BLANK}    /* skip */
983     \\n        BEGIN( INITIAL); lex_linepos = 0;
984     {LINENUM}  /* skip */
985     {FILENAME} setfilename();
986 }
987
988 <VALUE>{
989     \\n        lex_linenum++; lex_linepos = 0;

```

```

989 {BLANK}    /* skip */
990 {()}       return *yytext;
991 \\n        BEGIN( VALUE_STRING);
992 {ATOMCODE} atomcode(); return ATOM;
993 {ATOMSTR}  atom(); return ATOM;
994 }
995
996 <VALUE_STRING>{
997     \\n        BEGIN( VALUE);
998     {ATOMCODE} atomcode(); return ATOM;
999 }
1000
1001 {SATOMC}   atom(); return ATOM;
1002 }
1003
1004 {TRAFOID}   trafo(); return NAME;
1005 }
1006
1007 <*>{ANY}    return *yytext;
1008 }
1009 %%
1010
1011 void checkvaluemod( void) {
1012     if ( lex_value_switch) {
1013         lex_value_switch = NO;
1014         if ( YY_START == INITIAL) {
1015             BEGIN( VALUE);
1016             //printf( "\\nlex:VALUE-MODE,%d,%d\\n", VALUE, YY_START);
1017         }
1018         else {
1019             BEGIN( INITIAL);
1020             //puts( "INITIAL");
1021         }
1022     }
1023 }

```

lexparse.h

```

1024 #include <stdio.h>
1025 #include <stdlib.h>
1026 #include <string.h>
1027 #include "nop.h"
1028 #include "reprnode.h"
1029 #include "reprtraf.h"
1030 #include "reprpars.h"
1031 #include "nopalloc.h"
1032 #include "trafotab.h"
1033 #include "atomtab.h"
1034
1035 extern Bool lex_value_switch;
1036 extern long lex_linenum, lex_linepos;
1037 extern char *lex_filename;
1038
1039 int yylex( void);
1040
1041 Bool parser( void);
1042
1043 extern FILE *yyin;
1044 extern FILE *report;
1045 extern char repstr[];
1046
1047 typedef void ( *PRINTFUNC)( char *);
1048
1049 void printreport( char *s);
1050 void printerror( char *s);
1051 void printabort( char *s);

```

nop.y

```

1052 %{}
1053 #define YYERROR_VERBOSE
1054
1055 #include "lexparse.h"
1056 Bool lex_value_switch;

```



```

1057 long    lex_linenum, lex_linepos;
1058 char    *lex_filename;
1059
1060 #include "reprpars.h"
1061 TGroup   definitions;
1062
1063 #include "memnode.h"
1064
1065 #include <string.h>
1066 void yyerror( char* );
1067 static char errmsg[ 1000];
1068 %}
1069
1070 %union {
1071     Name      sv_name;
1072     TDef      sv_tdef;
1073     PTrf      sv_tref;
1074     Header    sv_header;
1075     Params    sv_paramlist;
1076     Args      sv_args;
1077     Value     sv_value;
1078     PNode     sv_node;
1079 }
1080
1081 %nonassoc      '?, '=', '&'
1082 %left          '.,'
1083 %left          '@'
1084
1085 %token <sv_name>      NAME
1086 %type <sv_tdef>       tdef
1087 %type <sv_tref>       tref sequence
1088 %type <sv_header>     header
1089 %type <sv_paramlist>  parameters paramlist
1090 %type <sv_args>       arguments arglist
1091
1092 %token <sv_name>      ATOM
1093 %type <sv_node>       node proppernode node_list
1094 %type <sv_value>     value internal_value
1095
1096 %%
1097
1098 program:
1099     tgroup
1100 ;
1101
1102 tgroup:
1103     tdef { add_def( &definitions, $1); }
1104 | tgroup tdef
1105 {
1106     if ( NOT add_def( &definitions, $2)) {
1107         strcpy( errmsg, "redefinition: ");
1108         strcat( errmsg, trafotable_search( $2.header.name));
1109         strcat( errmsg, "\n");
1110         sprintf( errmsg+strlen( errmsg), "%d", $2.header.params.nofparams);
1111         yyerror( errmsg);
1112     }
1113 }
1114
1115 ;
1116
1117 tdef:
1118     '+' header ':' tref
1119 {
1120     $$deftype = DEFT_USER;
1121     $$header = $2;
1122     $$select.tref = $4;
1123 };
1124
1125 header:
1126     NAME parameters { $$ = ( Header ) { $1, $2}; }
1127 ;
1128
1129 parameters:
1130     empty { $$ = ( Params ) { 0, NULL}; }
1131 | '(' paramlist ')' { $$ = $2; }

```

```

1132 ;
1133
1134 paramlist:
1135     NAME
1136 {
1137     $$ = ( Params ) { 0, NULL};
1138     add_param( &$$, $1);
1139 }
1140
1141 paramlist NAME
1142 {
1143     $$ = $1;
1144     if ( NOT add_param( &$$, $2)) {
1145         strcpy( errmsg, "duplicated parameter name: ");
1146         strcat( errmsg, trafotable_search( $2));
1147         yyerror( errmsg);
1148         YYABORT;
1149 }
1150 ;
1151
1152 tref:
1153     '\', value '\', { $$ = set_ref( $2); }
1154 | NAME arguments { $$ = name_ref( ( RefName ) { $1, -1, $2}); }
1155 | '*' { $$ = loop_ref(); }
1156 | tref ';' tref { $$ = seq_ref( $1, $3); }
1157 | '[' sequence ']' { $$ = $2; }
1158 | '@' tref { $$ = all_ref( $2); }
1159 | tref '?' tref { $$ = equ_ref( $1, $3); }
1160 | '=' tref '&' tref { $$ = equ_ref( $2, $4); }
1161 ;
1162
1163 arguments:
1164     empty { $$ = ( Args ) { 0, NULL}; }
1165 | '(' arglist ')' { $$ = $2; }
1166 ;
1167
1168 arglist:
1169     tref { $$ = ( Args ) { 0, NULL}; add_arg( &$$, $1); }
1170 | arglist tref { $$ = $1; add_arg( &$$, $2); }
1171 ;
1172
1173 sequence:
1174     empty { $$ = skip_ref(); }
1175 | tref { $$ = $1; }
1176 | sequence tref { $$ = seq_ref( $1, $2); }
1177 ;
1178
1179 empty:
1180 ;
1181
1182 /* values and nodes */
1183 value:
1184     { lex_value_switch = YES; }
1185     internal_value
1186     { lex_value_switch = YES; }
1187     $$ = $2;
1188 ;
1189
1190 internal_value:
1191     ATOM { $$ = setatom( $1); }
1192 | proppernode { $$ = setnode( $1); }
1193 ;
1194
1195 node:
1196     internal_value
1197 {
1198     if( isatom( $1))
1199         $$ = nodecreate( $1, NULL);
1200     else
1201         $$ = getnode( $1);
1202 };
1203
1204 proppernode:
1205     internal_value '(' node_list ')' { $$ = nodecreate( $1, $3); }
1206 ;

```



```

1207 node_list:
1208     empty                { $$ = NULL; }
1209 | node_list node        { $$ = nodelistappend( $1, $2); }
1210 ;
1211 %%
1212 void yyerror( char *s) {
1213     FILE *f;
1214     long line, pos, x;
1215     char c;
1216
1217     sprintf( repstr, "%s:%ld,%ld: %s\n",
1218             lex_filename, lex_linenum, lex_linepos, s);
1219     perror( repstr);
1220     if ( ( f = fopen( lex_filename, "r")) == NULL)
1221         return;
1222
1223     line = 1;
1224     pos = 0;
1225     while ( !feof( f) && ( line != lex_linenum))
1226         if ( getc( f) == '\n')
1227             line++;
1228
1229     pos = ftell( f);
1230     x = 0;
1231     while ( !feof( f) && ( ( c = getc( f)) != '\n'))
1232         sprintf( repstr+x++, "%c", c);
1233
1234     sprintf( repstr+x, "\n");
1235     perror( repstr);
1236
1237     fseek( f, pos, SEEK_SET);
1238     pos = 0;
1239     x = 0;
1240     while ( !feof( f) && ( ( c = getc( f)) != '\n'))
1241         if ( ++pos == lex_linepos)
1242             if ( ++pos == lex_linepos)
1243                 sprintf( repstr+x++, "%c", ' ');
1244             else
1245                 if ( c > ' ')
1246                     sprintf( repstr+x++, "%c", ' ');
1247                 else
1248                     sprintf( repstr+x++, "%c", c);
1249
1250     sprintf( repstr+x, "\n");
1251     perror( repstr);
1252
1253     fclose( f);
1254 }
1255
1256 Builtins builtin[ NOF_BUILTINS];
1257
1258 void builtin_setup( void) {
1259     builtin[ BC_exch]      .namecode = trafotable_insert( "exch");
1260     builtin[ BC_exch]      .arity = 0;
1261     builtin[ BC_cdn]       .namecode = trafotable_insert( "cdn");
1262     builtin[ BC_cdn]       .arity = 0;
1263     builtin[ BC_cup]       .namecode = trafotable_insert( "cup");
1264     builtin[ BC_cup]       .arity = 0;
1265     builtin[ BC_insert]    .namecode = trafotable_insert( "insert");
1266     builtin[ BC_insert]    .arity = 0;
1267     builtin[ BC_delete]    .namecode = trafotable_insert( "delete");
1268     builtin[ BC_delete]    .arity = 0;
1269     builtin[ BC_left]      .namecode = trafotable_insert( "left");
1270     builtin[ BC_left]      .arity = 0;
1271     builtin[ BC_right]     .namecode = trafotable_insert( "right");
1272     builtin[ BC_right]     .arity = 0;
1273     builtin[ BC_split]     .namecode = trafotable_insert( "split");
1274     builtin[ BC_split]     .arity = 0;
1275     builtin[ BC_replace]   .namecode = trafotable_insert( "replace");
1276     builtin[ BC_replace]   .arity = 0;
1277     builtin[ BC_skip]      .namecode = trafotable_insert( "skip");
1278     builtin[ BC_skip]      .arity = 0;
1279     builtin[ BC_seq]       .namecode = trafotable_insert( "seq");
1280     builtin[ BC_seq]       .arity = 2;
1281

```

```

1282     builtin[ BC_equ]       .namecode = trafotable_insert( "equ");
1283     builtin[ BC_equ]       .arity = 2;
1284     builtin[ BC_all]       .namecode = trafotable_insert( "all");
1285     builtin[ BC_all]       .arity = 1;
1286     builtin[ BC_trace]     .namecode = trafotable_insert( "trace");
1287     builtin[ BC_trace]     .arity = 0;
1288     builtin[ BC_trace0]    .namecode = trafotable_insert( "trace0");
1289     builtin[ BC_trace0]    .arity = 0;
1290     builtin[ BC_trace1]    .namecode = trafotable_insert( "trace1");
1291     builtin[ BC_trace1]    .arity = 0;
1292     builtin[ BC_dump]      .namecode = trafotable_insert( "dump");
1293     builtin[ BC_dump]      .arity = 0;
1294
1295     bin_def( &definitions, BC_exch);
1296     bin_def( &definitions, BC_cdn);
1297     bin_def( &definitions, BC_cup);
1298     bin_def( &definitions, BC_insert);
1299     bin_def( &definitions, BC_delete);
1300     bin_def( &definitions, BC_left);
1301     bin_def( &definitions, BC_right);
1302     bin_def( &definitions, BC_split);
1303     bin_def( &definitions, BC_replace);
1304     bin_def( &definitions, BC_skip);
1305     bin_def( &definitions, BC_seq);
1306     bin_def( &definitions, BC_equ);
1307     bin_def( &definitions, BC_all);
1308     bin_def( &definitions, BC_trace);
1309     bin_def( &definitions, BC_trace0);
1310     bin_def( &definitions, BC_trace1);
1311     bin_def( &definitions, BC_dump);
1312 }
1313
1314 Bool parser( void) {
1315
1316     lex_linenum = 1;
1317     lex_linepos = 0;
1318     lex_filename = NULL;
1319     lex_value_switch = NO;
1320
1321     atomtable_init();
1322     trafotable_init();
1323
1324     definitions = ( TGroup) { 0, NULL};
1325     builtin_setup();
1326
1327     //yydebug = 1;
1328
1329     return yyparse();
1330 }

```

strtable.h

```

1331 #ifndef strtable_included
1332 #define strtable_included
1333
1334 #include "nop.h"
1335
1336 typedef char*   StrTableString;
1337 typedef Uint    StrTableCode;
1338
1339 typedef void*   StrTableId;
1340 typedef void    ( *FStrTableEnum)( StrTableCode code, StrTableString str);
1341
1342 StrTableId strtable_init( Uint startcode, Uint sizeincrement);
1343 void strtable_destroy( StrTableId id);
1344
1345 Bool strtable_lookup( StrTableId id, StrTableString str, StrTableCode *pcode);
1346 Bool strtable_insert( StrTableId id, StrTableString str, StrTableCode *pcode);
1347 Bool strtable_add( StrTableId id, StrTableString str, StrTableCode *pcode);
1348 Bool strtable_search( StrTableId id, StrTableCode code, StrTableString *str);
1349 void strtable_enum( StrTableId id, FStrTableEnum enumfunc);
1350
1351 #endif

```

strtable.c


```

1352 #include <stdio.h>
1353 #include <string.h>
1354 #include "strtable.h"
1355 #include "nopalloc.h"
1356
1357 typedef struct {
1358     Uint      sizeinc;
1359     Uint      maxnofitems;
1360     Uint      nofitems;
1361     Uint      startcode;
1362     StrTableString* strttable;
1363 } StrTableDesc;
1364
1365 typedef StrTableDesc* PStrTableDesc;
1366
1367 StrTableId strttable_init( Uint startcode, Uint sizeincrement) {
1368     PStrTableDesc pd;
1369
1370     pd = nopalloc( sizeof *pd);
1371     pd->sizeinc = sizeincrement;
1372     pd->maxnofitems = 0;
1373     pd->nofitems = 0;
1374     pd->startcode = startcode;
1375     pd->strttable = NULL;
1376
1377     return pd;
1378 }
1379
1380 void strttable_destroy( StrTableId id) {
1381     PStrTableDesc pd = id;
1382     Uint i;
1383
1384     for ( i = 0; i < pd->nofitems; i++)
1385         free( pd->strttable+i);
1386
1387     free( pd);
1388 }
1389
1390 // str->code
1391 //
1392 Bool strttable_lookup( StrTableId id, StrTableString str, StrTableCode *pcode) {
1393     PStrTableDesc pd = id;
1394     Uint i;
1395
1396     for ( i = 0; i < pd->nofitems; i++)
1397         if ( !strcmp( str, ( pd->strttable)[ i])) {
1398             *pcode = i + pd->startcode;
1399             return YES;
1400         }
1401
1402     return NO;
1403 }
1404
1405 static void checkalloc( PStrTableDesc pd) {
1406     if ( pd->nofitems == pd->maxnofitems) {
1407         pd->maxnofitems += pd->sizeinc;
1408         pd->strttable =
1409             nprealloc( pd->strttable, pd->maxnofitems * sizeof ( StrTableString));
1410     }
1411 }
1412
1413 Bool strttable_insert( StrTableId id, StrTableString str, StrTableCode *pcode) {
1414     PStrTableDesc pd = id;
1415
1416     if ( strttable_lookup( pd, str, pcode))
1417         return NO;
1418     else {
1419         StrTableString s;
1420
1421         checkalloc( pd);
1422         s = nopalloc( strlen( str)+1);
1423         strcpy( s, str);
1424         pd->strttable[ pd->nofitems] = s;
1425         *pcode = pd->startcode + pd->nofitems++;
1426         return YES;

```

```

1427     }
1428 }
1429
1430 // str->code
1431 // str: null terminated string
1432 // str not copied, it should not be accessed by caller after return
1433 //
1434 Bool strttable_add( StrTableId id, StrTableString str, StrTableCode *pcode) {
1435     PStrTableDesc pd = id;
1436
1437     if ( strttable_lookup( pd, str, pcode)) {
1438         free( str);
1439         return NO;
1440     }
1441     else {
1442         checkalloc( pd);
1443         pd->strttable[ pd->nofitems] = str;
1444         *pcode = pd->startcode + pd->nofitems++;
1445         return YES;
1446     }
1447 }
1448
1449 // code->str
1450 //
1451 Bool strttable_search( StrTableId id, StrTableCode code, StrTableString *str) {
1452     PStrTableDesc pd = id;
1453     StrTableCode idx;
1454
1455     if ( ( idx = code - pd->startcode) < pd->nofitems) {
1456         *str = pd->strttable[ idx];
1457         return YES;
1458     }
1459     else {
1460         *str = NULL;
1461         return NO;
1462     }
1463 }
1464
1465 void strttable_enum( StrTableId id, FStrTableEnum enumfunc) {
1466     PStrTableDesc pd = id;
1467     Uint i;
1468
1469     for ( i = 0; i < pd->nofitems; i++)
1470         enumfunc( pd->startcode+i, pd->strttable[ i]);
1471 }

```

trafotab.h

```

1472 #include "strtable.h"
1473
1474 void      trafotable_init( void);
1475 void      trafotable_destroy( void);
1476 StrTableCode trafotable_lookup( StrTableString);
1477 StrTableCode trafotable_insert( StrTableString);
1478 StrTableString trafotable_search( StrTableCode);
1479 void      trafotable_list( void);

```

trafotab.c

```

1480 #include <stdio.h>
1481 #include "trafotab.h"
1482 #include "lexparse.h"
1483
1484 #define TRAFOCODE_START 0
1485 #define TRAFOTABLE_INCR 1000
1486
1487 static StrTableId trafo_table;
1488
1489 void trafotable_init( void) {
1490     trafo_table = strttable_init( TRAFOCODE_START, TRAFOTABLE_INCR);
1491 }
1492
1493 void trafotable_destroy( void) {
1494     strttable_destroy( trafo_table);

```



```

1495 }
1496
1497 StrTableCode trafotable_lookup( StrTableString str) {
1498     StrTableCode code;
1499
1500     if ( !strtable_lookup( trafo_table, str, &code)) {
1501         fprintf( stderr, "abort: trafotable_lookup\n");
1502         abort();
1503     }
1504
1505     return code;
1506 }
1507
1508 StrTableCode trafotable_insert( StrTableString str) {
1509     StrTableCode code;
1510
1511     strtable_insert( trafo_table, str, &code);
1512     return code;
1513 }
1514
1515 StrTableString trafotable_search( StrTableCode code) {
1516     StrTableString str;
1517
1518     if ( !strtable_search( trafo_table, code, &str)) {
1519         fprintf( stderr, "abort: trafotable_search\n");
1520         abort();
1521     }
1522     return str;
1523 }
1524
1525 static void trafonameprint( StrTableCode code, StrTableString str) {
1526     sprintf( repstr, "%5d: \"%s\"\n", code, str);
1527     printreport( repstr);
1528 }
1529
1530 void trafotable_list( void) {
1531     strtable_enum( trafo_table, trafonameprint);
1532 }

```

```

atomtab.h

```

```

1533 #include "strtable.h"
1534
1535 void          atomtable_init( void);
1536 void          atomtable_destroy( void);
1537 StrTableCode  atomtable_lookup( StrTableString);
1538 StrTableCode  atomtable_insert( StrTableString);
1539 StrTableString atomtable_search( StrTableCode);
1540 void          atomtable_list( void);

```

```

atomtab.c

```

```

1541 #include <stdio.h>
1542 #include "atomtab.h"
1543 #include "lexparse.h"
1544
1545 #define ATOMCODE_START 256
1546 #define ATOMTABLE_INCR 1000
1547
1548 static StrTableId  atom_table;
1549 static char        charvector[ ATOMCODE_START][ 2];
1550
1551 void atomtable_init( void) {
1552     Uint i;
1553
1554     atom_table = strtable_init( ATOMCODE_START, ATOMTABLE_INCR);
1555     for ( i = 0; i < ATOMCODE_START; i++) {
1556         charvector[ i][ 0] = i;
1557         charvector[ i][ 1] = '\0';
1558     }
1559 }
1560
1561 void atomtable_destroy( void) {
1562     strtable_destroy( atom_table);

```

```

1563 }
1564
1565 StrTableCode atomtable_lookup( StrTableString str) {
1566     StrTableCode code;
1567
1568     if ( ( str[ 0] != '\0') && ( str[ 1] == '\0'))
1569         code = str[ 0];
1570     else
1571         if ( !strtable_lookup( atom_table, str, &code)) {
1572             fprintf( stderr, "atomtable_lookup\n");
1573             abort();
1574         }
1575     return code;
1576 }
1577
1578 StrTableCode atomtable_insert( StrTableString str) {
1579     StrTableCode code;
1580
1581     if ( ( str[ 0] != '\0') && ( str[ 1] == '\0'))
1582         code = str[ 0];
1583     else
1584         strtable_insert( atom_table, str, &code);
1585     return code;
1586 }
1587
1588 StrTableString atomtable_search( StrTableCode code) {
1589     StrTableString str;
1590
1591     if ( code < ATOMCODE_START)
1592         return charvector[ code];
1593
1594     if ( !strtable_search( atom_table, code, &str)) {
1595         fprintf( stderr, "atomtable_search\n");
1596         abort();
1597     }
1598     return str;
1599 }
1600
1601 static void atomprint( StrTableCode code, StrTableString str) {
1602     sprintf( repstr, "%5d: \"%s\"\n", code, str);
1603     printreport( repstr);
1604 }
1605
1606 void atomtable_list( void) {
1607     strtable_enum( atom_table, atomprint);
1608 }

```

```

noptrav.h

```

```

1609 #ifndef noptrav_included
1610 #define noptrav_included
1611
1612 #include <stdio.h>
1613 #include "nop.h"
1614 #include "atomtab.h"
1615 #include "trafotab.h"
1616 #include "reprpars.h"
1617 #include "reprtraf.h"
1618 #include "lexparse.h"
1619
1620 typedef void ( *NodeVoidFunc)();
1621 void nodeprint( PNode, FILE *, Bool simple);
1622 void valueprint( Value, FILE *);
1623
1624 void printref( PRef ptr, Uint indent, PRINTFUNC pf);
1625 void printdef( PDef pdef, PRINTFUNC pf);
1626 void printdefs( void);
1627 void printtrafo( PTrafo ptr, PRINTFUNC pf);
1628 void printprog( void);
1629 //void printbin( Builtin);
1630 //void scopenameprint( PDefBlock, char*);
1631
1632 #endif

```

```

noptrav.c

```



```

1633 #include "noptrav.h"
1634
1635 static PRINTFUNC printfunc;
1636
1637 void nodeprint( PNode pnode, FILE *f, Bool simple) {
1638     valueprint( pnode->value, f);
1639
1640     if ( NOT simple)
1641         fprintf( f, "(");
1642
1643     if ( ( pnode->pfirstsub != NULL)) {
1644         Bool xsimple;
1645         register PNode p;
1646
1647         p = pnode->pfirstsub;
1648         xsimple = YES;
1649         do {
1650             xsimple = xsimple AND isatom( p->value) AND ( p->pfirstsub == NULL);
1651             p = p->pnext;
1652         } While ( p != pnode->pfirstsub);
1653
1654         p = pnode->pfirstsub;
1655         do {
1656             nodeprint( p, f, xsimple);
1657             p = p->pnext;
1658             if ( p != pnode->pfirstsub)
1659                 fprintf( f, " ");
1660         } while ( p != pnode->pfirstsub);
1661     }
1662
1663     if ( NOT simple)
1664         fprintf( f, ")");
1665 }
1666
1667 void valueprint( Value value, FILE *f) {
1668     StrTableString atomstr;
1669
1670     if ( isatom( value)) {
1671         atomstr = atomtable_search( getatom( value));
1672         if ( atomstr[ 1] == '0')
1673             switch ( atomstr[ 0]) {
1674                 case 0 ... 31:
1675                     fprintf( f, "%02x", atomstr[ 0]);
1676                     break;
1677                 case '#':
1678                     fprintf( f, "##");
1679                     break;
1680                 default:
1681                     fprintf( f, "%c", atomstr[ 0]);
1682                     break;
1683             }
1684     }
1685     else
1686         fprintf( f, atomstr);
1687 }
1688
1689 else
1690     nodeprint( getnode( value), f, NO);
1691 }
1692
1693 void printlref( TRef* pref, Uint indent) {
1694     Uint i, x;
1695
1696     if ( indent > 0) {
1697         sprintf( repstr, "\n%c", indent, ' ');
1698         printfunc( repstr);
1699     }
1700
1701     switch ( pref->reftype) {
1702     case REFT_SET:
1703         printfunc( "set(");
1704         valueprint( pref->select.value, report);
1705         printfunc( ")");
1706         break;
1707     case REFT_NAME:
1708         printfunc( trafotable_search( pref->select.refname.name));

```

```

1708     if ( pref->select.refname.localidx >= 0) {
1709         sprintf( repstr, "%d", pref->select.refname.localidx);
1710         printfunc( repstr);
1711     }
1712     if ( ( x = pref->select.refname.args.nofargs) != 0) {
1713         sprintf( repstr, "%d", pref->select.refname.args.nofargs);
1714         printfunc( repstr);
1715         for ( i = 0; i < x; i++)
1716             printlref( pref->select.refname.args.argv[ i], indent+1);
1717     }
1718     break;
1719 case REFT_LOOP:
1720     printfunc( "*");
1721     break;
1722 default:
1723     printabort( "printlref: ??? REFERENCE ERROR ???");
1724     break;
1725 }
1726
1727 void printref( TRef* pref, Uint indent, PRINTFUNC pf) {
1728     printfunc = pf;
1729     printlref( pref, indent);
1730 }
1731
1732 void printldef( PDef pd, Uint indent) {
1733     Uint i, x;
1734
1735     if ( indent > 0) {
1736         sprintf( repstr, "\n%c", indent, ' ');
1737         printfunc( repstr);
1738     }
1739     sprintf( repstr, "+s", trafotable_search( pd->header.name));
1740     printfunc( repstr);
1741
1742     switch ( pd->deftype) {
1743     case DEFT_BUILTIN:
1744         sprintf( repstr, "%d : %d",
1745             pd->header.params.nofparams, pd->select.bincode);
1746         printfunc( repstr);
1747         break;
1748     case DEFT_USER:
1749         if ( ( x = pd->header.params.nofparams) != 0) {
1750             printfunc( "(");
1751             for ( i = 0; i < x; i++) {
1752                 sprintf( repstr, "%d:%s-%d", i,
1753                     trafotable_search( pd->header.params.paramv[ i].name),
1754                     pd->header.params.paramv[ i].refcnt);
1755                 printfunc( repstr);
1756             }
1757             printfunc( ")");
1758         }
1759         printlref( pd->select.tref, indent+1);
1760         break;
1761     default:
1762         printabort( "printldef:????????????????????");
1763         break;
1764     }
1765     printfunc( "\n");
1766 }
1767
1768 void printdefs( void) {
1769     Uint i;
1770
1771     printfunc = printreport;
1772     for ( i = 0; i < definitions.nofdefs; i++)
1773         printldef( definitions.defv[ i], 0);
1774 }
1775
1776 void printdef( PDef ptdef, PRINTFUNC pf) {
1777     printfunc = pf;
1778     printldef( ptdef, 0);
1779 }
1780
1781 void printbin( BinCode bc) {

```



```

1783 switch( bc ) {
1784 case BC_exch:
1785     printf( "exch\n");
1786     break;
1787 case BC_cdn:
1788     printf( "cdn\n");
1789     break;
1790 case BC_cup:
1791     printf( "cup\n");
1792     break;
1793 case BC_insert:
1794     printf( "insert\n");
1795     break;
1796 case BC_delete:
1797     printf( "delete\n");
1798     break;
1799 case BC_left:
1800     printf( "left\n");
1801     break;
1802 case BC_right:
1803     printf( "right\n");
1804     break;
1805 case BC_split:
1806     printf( "split\n");
1807     break;
1808 case BC_replace:
1809     printf( "replace\n");
1810     break;
1811 case BC_skip:
1812     printf( "skip\n");
1813     break;
1814 case BC_seq:
1815     printf( "seq\n");
1816     break;
1817 case BC_equ:
1818     printf( "equ\n");
1819     break;
1820 case BC_all:
1821     printf( "all\n");
1822     break;
1823 default:
1824     printabort( "printbin: invalid builtin code\n");
1825     break;
1826 }
1827 }
1828
1829 void printitrafo( PTrafo ptrafa, Uint indent, Bool seqflag) {
1830     void doindent( void) {
1831         if ( indent > 0 ) {
1832             sprintf( repstr, "%*c", indent, ' ');
1833             printf( repstr);
1834         }
1835     }
1836
1837     Uint i;
1838
1839     if ( ptrafa == NULL ) {
1840         printf( "---");
1841         return;
1842     }
1843
1844     if ( ptrafa->refid > 0 ) {
1845         doindent();
1846         if ( ptrafa->ready ) {
1847             sprintf( repstr, ">%d\n", ptrafa->refid);
1848             printf( repstr);
1849             indent++;
1850         }
1851         else {
1852             sprintf( repstr, "%*d\n", ptrafa->refid);
1853             printf( repstr);
1854             return;
1855         }
1856     }
1857 }

```

```

1858 switch ( ptrafa->type) {
1859 case TT_SET:
1860     doindent();
1861     printf( "set( ");
1862     valueprint( ptrafa->select.value, report);
1863     printf( ")\n");
1864     break;
1865 case TT_BIN: {
1866     BinCode bc;
1867     Bool incident;
1868
1869     bc = ptrafa->select.bin.bc;
1870     if ( ( incident = ( bc != BC_seq) || ( NOT seqflag) ) ) {
1871         doindent();
1872         sprintf( repstr, "%s\n", trafotable_search( builtin[ bc].namecode));
1873         printf( repstr);
1874     }
1875     for ( i = 0; i < builtin[ bc].arity; i++)
1876         printitrafo( ptrafa->select.bin.args[ i],
1877                     incident ? indent+i: indent,
1878                     bc == BC_seq);
1879     break;
1880 }
1881 case TT_NAME:
1882     doindent();
1883     sprintf( repstr, "%s\n", trafotable_search( ptrafa->select.name.name));
1884     printf( repstr);
1885     for ( i = 0; i < ptrafa->select.name.nofargs; i++)
1886         printitrafo( ptrafa->select.name.args[ i], indent+1, NO);
1887     break;
1888 case TT_LOOP:
1889     doindent();
1890     if ( ( i = ptrafa->select.loop->refid) > 0 ) {
1891         sprintf( repstr, "%d\n", i);
1892         printf( repstr);
1893     }
1894     break;
1895 default:
1896     doindent();
1897     printabort( "printitrafo: unknown trafo type\n");
1898     break;
1899 }
1900
1901 ptrafa->ready = NO;
1902
1903 void printtrafo( PTrafo ptrafa, PRINTFUNC pf) {
1904     printf( repstr);
1905     printf( repstr);
1906     printf( repstr);
1907     printf( repstr);
1908     printf( repstr);
1909 }
1910
1911 void printprog( void) {
1912     printtrafo( program, printreport);
1913 }

```

reprpars.h

```

1914 #ifndef reprpars_included
1915 #define reprpars_included
1916
1917 #include "nop.h"
1918 #include "reprnode.h"
1919 #include "reprtraf.h"
1920 #include "strtable.h"
1921
1922 typedef struct TGroup
1923 typedef TGroup*
1924 typedef struct TDef
1925 typedef TDef*
1926 typedef struct TRef
1927 typedef TRef*
1928 typedef struct Header
1929 typedef struct Param

```

```

TGroup;
PTGroup;
TDef;
PTDef;
TRef;
PTRef;
Header;
Param;

```



```

1930 typedef Param*      PParam;
1931 typedef struct Params Params;
1932 typedef Params*      PParams;
1933 typedef struct RefName RefName;
1934 typedef struct Args Args;
1935 typedef Args*      PArgs;
1936
1937 struct Args {
1938     UInt      nofargs;
1939     PRef      *argv;
1940 };
1941
1942 struct RefName {
1943     Name      name;
1944     int      localidx;    // <0 if name is global
1945     Args      args;
1946 };
1947
1948 typedef enum {
1949     REFT_SET,
1950     REFT_NAME,
1951     REFT_LOOP
1952 } RefType;
1953
1954 struct TRef {
1955     RefType      reftype;
1956     union {
1957         Value      value;    // REFT_SET
1958         RefName     refname;  // REFT_NAME
1959         // nothing    REFT_LOOP
1960     }
1961     select;
1962 };
1963
1964 struct Param {
1965     Name      name;
1966     UInt      refcnt;
1967 };
1968
1969 struct Params {
1970     UInt      nofparams;
1971     PParam     paramv;
1972 };
1973
1974 struct Header {
1975     Name      name;
1976     Params     params;
1977 };
1978
1979 struct TDef {
1980     enum {
1981         DEFT_BUILTIN,
1982         DEFT_USER
1983     }      deftype;
1984     Header      header;
1985     //PTGroup     subdefs;
1986     union {
1987         BinCode     bincode;    // DEFT_BUILTIN
1988         PRef         tref;       // DEFT_USER
1989     }
1990     select;
1991 };
1992
1993 struct TGroup {
1994     UInt      nofdefs;
1995     PTDef      *defv;
1996 };
1997
1998 extern TGroup     definitions;
1999 extern Name      builtin_namecode[ NOF_BUILTINS];
2000
2001 PRef set_ref( Value value);
2002 PRef name_ref( RefName ref);
2003 PRef loop_ref( void);
2004 PRef skip_ref( void);
2005 Bool add_param( PParams pparams, Name param);
2006 void add_arg( PArgs pargs, PRef ptref);

```

```

2005 PRef seq_ref( PRef pseq, PRef pref);
2006 PRef equ_ref( PRef true, PRef false);
2007 PRef all_ref( PRef ptref);
2008 Bool add_def( PTGroup pg, TDef tdef);
2009 Bool bin_def( PTGroup pg, BinCode bincode);
2010
2011 #endif

```

reprpars.c

```

2012 #include "reprpars.h"
2013 #include "reprtraf.h"
2014 #include "nopalloc.h"
2015
2016 static PRef newref( RefType reftype) {
2017     PRef ptref;
2018
2019     ptref = nopalloc( sizeof *ptref);
2020     ptref->reftype = reftype;
2021     return ptref;
2022 }
2023
2024 PRef set_ref( Value value) {
2025     PRef ptref;
2026
2027     ptref = newref( REFT_SET);
2028     ptref->select.value = value;
2029     return ptref;
2030 }
2031
2032 PRef name_ref( RefName ref) {
2033     PRef ptref;
2034
2035     ptref = newref( REFT_NAME);
2036     ptref->select.refname = ref;
2037     return ptref;
2038 }
2039
2040 PRef loop_ref( void) {
2041     return newref( REFT_LOOP);
2042 }
2043
2044 PRef skip_ref( void) {
2045     return name_ref( ( RefName) { builtin[ BC_skip].namecode, -1, { 0, NULL}});
2046 }
2047
2048 PRef seq_ref( PRef pseq, PRef pref) {
2049     Args args = ( Args) { 0, NULL};
2050
2051     add_arg( &args, pseq);
2052     add_arg( &args, pref);
2053     return name_ref( ( RefName) { builtin[ BC_seq].namecode, -1, args});
2054 }
2055
2056 PRef equ_ref( PRef true, PRef false) {
2057     Args args = ( Args) { 0, NULL};
2058
2059     add_arg( &args, true);
2060     add_arg( &args, false);
2061     return name_ref( ( RefName) { builtin[ BC_equ].namecode, -1, args});
2062 }
2063
2064 PRef all_ref( PRef ptref) {
2065     Args args = ( Args) { 0, NULL};
2066
2067     add_arg( &args, ptref);
2068     return name_ref( ( RefName) { builtin[ BC_all].namecode, -1, args});
2069 }
2070
2071 Bool add_param( PParams pparams, Name param) {
2072     const UInt      increment = 10;
2073     UInt            i, maxparams;
2074
2075     if ( pparams->nofparams % increment == 0) {
2076         maxparams = pparams->nofparams + increment;

```



```

2077     pparams->paramv =
2078     noprealloc( pparams->paramv, maxparams * sizeof ( Param));
2079 }
2080
2081 pparams->paramv[ pparams->nofparams++] = ( Param) { param, 0};
2082
2083 for ( i = 0; i < pparams->nofparams-1; i++)
2084     if ( param == pparams->paramv[ i].name)
2085         return NO;
2086
2087 return YES;
2088 }
2089
2090 void add_arg( PArgs pargs, PTrRef ptref) {
2091     const UInt    increment = 10;
2092     UInt          maxargs;
2093
2094     if ( pargs->nofargs % increment == 0) {
2095         maxargs = pargs->nofargs + increment;
2096         pargs->argv = noprealloc( pargs->argv, maxargs * sizeof ( PTrRef));
2097     }
2098     pargs->argv[ pargs->nofargs++] = ptref;
2099 }
2100
2101 Bool add_def( PTGroup pg, TDef tdef) {
2102
2103     Bool cmphead( Header h1, Header h2) {
2104         return
2105             ( h1.name == h2.name) &&
2106             ( h1.params.nofparams == h2.params.nofparams);
2107     }
2108
2109     const UInt    increment = 10;
2110     UInt          i, maxdefs;
2111     PTDef         ptdef;
2112
2113     for ( i = 0; i < pg->nofdefs; i++)
2114         if ( cmphead( pg->defv[ i]->header, tdef.header))
2115             return NO;
2116
2117     if ( pg->nofdefs % increment == 0) {
2118         maxdefs = pg->nofdefs + increment;
2119         pg->defv = noprealloc( pg->defv, maxdefs * sizeof ( PTDef));
2120     }
2121
2122     ptdef = nopalloc( sizeof ( TDef));
2123     *ptdef = tdef;
2124
2125     pg->defv[ pg->nofdefs++] = ptdef;
2126     return YES;
2127 }
2128
2129 Bool bin_def( PTGroup pg, BinCode bincode) {
2130     TDef tdef;
2131
2132     tdef.deftype = DEFT_BUILTIN;
2133     tdef.header.name = builtin[ bincode].namecode;
2134     tdef.header.params = ( Params) { builtin[ bincode].arity, NULL};
2135     tdef.select.bincode = bincode;
2136     return add_def( pg, tdef);
2137 }
2138 }

```

C.5 Executor

```
nopexec.h
```

```
2139 void execute( void);
```

```
nopexec.c
```

```

2140 #include <stdio.h>
2141
2142 #include "nop.h"
2143
2144 #include "memexec.h"
2145 #include "memstack.h"
2146 #include "memnode.h"
2147 #include "trafotab.h"
2148 #include "atomtab.h"
2149 #include "nopexec.h"
2150 #include "reprtraf.h"
2151 #include "noptrav.h"
2152
2153 static void discard_value( Value value);
2154 static void discard_node( PNode pnode);
2155
2156 static void discard_nodelist( PNode pnode) {
2157     if ( pnode != NULL) {
2158         PNode p, q;
2159
2160         q = pnode;
2161         do {
2162             q = ( p = q)->pnext;
2163             discard_node( p);
2164         } while ( q != pnode);
2165     }
2166 }
2167
2168 static void discard_node( PNode pnode) {
2169     if ( pnode != NULL) {
2170         discard_value( pnode->value);
2171         discard_nodelist( pnode->pfirstsub);
2172         nodefree( pnode);
2173     }
2174 }
2175
2176 static void discard_value( Value value) {
2177     if ( isnode( value))
2178         discard_node( getnode( value));
2179 }
2180
2181 static PNode copy_node( PNode pnode, UInt *time);
2182 static Value copy_value( Value value, UInt *time);
2183
2184 static void linkup( PNode new, PNode rest) {
2185     new->pnext = rest;
2186     new->pprev = rest->pprev;
2187     rest->pprev->pnext = new;
2188     rest->pprev = new;
2189 }
2190
2191 static PNode copy_nodelist( PNode psubs, UInt *time) {
2192     PNode pcopy;
2193     UInt t, tx;
2194
2195     t = 0;
2196     if ( ( pcopy = psubs) != NULL) {
2197         PNode p, pcprev, pcnext;
2198
2199         p = psubs;
2200         pcopy = pcprev = copy_node( p, &tx);
2201         t = tx;
2202         while ( ( p = p->pprev) != psubs) {
2203             pcnext = copy_node( p, &tx);
2204             t += tx;
2205             linkup( pcnext, pcprev);
2206             pcprev = pcnext;
2207         }
2208     }
2209
2210     *time = t;
2211     return pcopy;
2212 }
2213
2214 static PNode copy_node( PNode pnode, UInt *time) {

```



```

2215 Value value;
2216 PNode psubs;
2217 Uint t1, t2;
2218
2219 value = copy_value( pnode->value, &t1);
2220 psubs = copy_nodelist( pnode->pfirstsub, &t2);
2221
2222 *time = t1+t2+1;
2223 return nodeallocfill( value, psubs);
2224 }
2225
2226 static Value copy_value( Value original, Uint *time) {
2227   if ( isatom( original)) {
2228     *time = 0;
2229     return original;
2230   }
2231   else
2232     return setnode( copy_node( getnode( original), time));
2233 }
2234
2235 static Bool cmp_node( PNode p1, PNode p2, Uint *time);
2236 static Bool cmp_value( Value v1, Value v2, Uint *time);
2237
2238 static Bool cmp_nodelist( PNode p1, PNode p2, Uint *time) {
2239   Bool result;
2240   Uint t, tx;
2241
2242   if ( ( p1 == NULL) || ( p2 == NULL)) {
2243     t = 1;
2244     result = ( p1 == p2);
2245   }
2246   else {
2247     PNode pt1, pt2;
2248     Bool x1, x2, match;
2249
2250     t = 0;
2251     pt1 = p1;
2252     pt2 = p2;
2253     match = YES;
2254     do {
2255       match = cmp_node( pt1, pt2, &tx);
2256       t += tx;
2257       x1 = ( pt1 == pt1->pnext) == p1;
2258       x2 = ( pt2 == pt2->pnext) == p2;
2259     } while ( match AND NOT ( x1 OR x2));
2260
2261     result = ( match AND x1 AND x2);
2262   }
2263
2264   *time = t;
2265   return result;
2266 }
2267
2268 static Bool cmp_node( PNode p1, PNode p2, Uint *time) {
2269   Uint t1, t2;
2270   Bool result;
2271
2272   t1 = t2 = 0;
2273   result = (
2274     cmp_value( p1->value, p2->value, &t1) AND
2275     cmp_nodelist( p1->pfirstsub, p2->pfirstsub, &t2)
2276   );
2277   *time = t1+t2+1;
2278   return result;
2279 }
2280
2281 static Bool cmp_value( Value value1, Value value2, Uint *time) {
2282   Bool result;
2283   Uint t;
2284
2285   t = 0;
2286   result = NO;
2287   if ( isatom( value1) AND
2288       isatom( value2) AND
2289       ( getatom( value1) == getatom( value2))

```

```

2290   )
2291   result = YES;
2292
2293   if ( isnode( value1) AND
2294       isnode( value2) AND
2295       cmp_node( getnode( value1), getnode( value2), &t)
2296   )
2297     result = YES;
2298
2299   *time = t+1;
2300   return result;
2301 }
2302
2303 static Bool traceflag;
2304 static PExecUnit peu;
2305
2306 typedef Uint ( *EXEFCUNC)( PTrafo *);
2307 static EXEFCUNC execfunc[ NOF_BUILTINS];
2308
2309 static Uint exec_exch( PTrafo *dummy) {
2310   if ( peu->pnode->pfirstsub != NULL) {
2311     Value v;
2312
2313     v = peu->pnode->pfirstsub->value;
2314     peu->pnode->pfirstsub->value = peu->pnode->value;
2315     peu->pnode->value = v;
2316   }
2317   return 1;
2318 }
2319
2320 static Uint exec_cdn( PTrafo *dummy) {
2321   Uint time = 1;
2322
2323   if ( peu->pnode->pfirstsub != NULL) {
2324     discard_value( peu->pnode->pfirstsub->value);
2325     peu->pnode->pfirstsub->value = copy_value( peu->pnode->value, &time);
2326   }
2327   return time;
2328 }
2329
2330 static Uint exec_cup( PTrafo *dummy) {
2331   Uint time = 1;
2332
2333   if ( peu->pnode->pfirstsub != NULL) {
2334     discard_value( peu->pnode->value);
2335     peu->pnode->value = copy_value( peu->pnode->pfirstsub->value, &time);
2336   }
2337   return time;
2338 }
2339
2340 static Uint exec_insert( PTrafo *dummy) {
2341   PNode p;
2342
2343   p = nodeallocfill( setatom( 0), NULL);
2344   if ( peu->pnode->pfirstsub != NULL)
2345     linkup( p, peu->pnode->pfirstsub);
2346   peu->pnode->pfirstsub = p;
2347   return 1;
2348 }
2349
2350 static Uint exec_delete( PTrafo *dummy) {
2351   PNode p, new;
2352
2353   if ( ( p = peu->pnode->pfirstsub) != NULL) {
2354     new = p->pnext;
2355     if ( new == p)
2356       new = NULL;
2357     else {
2358       p->pprev->pnext = new;
2359       new->pprev = p->pprev;
2360     }
2361     discard_node( p);
2362     peu->pnode->pfirstsub = new;
2363   }
2364   return 1;

```



```

2365 }
2366 static Uint exec_left( PTrafo *dummy) {
2367     if ( peu->pnode->pfirstsub != NULL)
2368         peu->pnode->pfirstsub = peu->pnode->pfirstsub->pprev;
2369     return 1;
2370 }
2371 }
2372 static Uint exec_right( PTrafo *dummy) {
2373     if ( peu->pnode->pfirstsub != NULL)
2374         peu->pnode->pfirstsub = peu->pnode->pfirstsub->pnext;
2375     return 1;
2376 }
2377 }
2378 static Uint exec_split( PTrafo *dummy) {
2379     PNode p;
2380     p = nodeallocfill( peu->pnode->value, peu->pnode->pfirstsub);
2381     peu->pnode->value = setnode( p);
2382     peu->pnode->pfirstsub = NULL;
2383     return 1;
2384 }
2385 }
2386 static Uint exec_replace( PTrafo *dummy) {
2387     discard_nodelist( peu->pnode->pfirstsub);
2388     if ( isnode( peu->pnode->value)) {
2389         PNode p;
2390         p = getnode( peu->pnode->value);
2391         peu->pnode->value = p->value;
2392         peu->pnode->pfirstsub = p->pfirstsub;
2393         p->value = setatom( 0);
2394         p->pfirstsub = NULL;
2395         discard_node( p);
2396     }
2397     else
2398         peu->pnode->pfirstsub = NULL;
2399     return 1;
2400 }
2401 static Uint exec_skip( PTrafo *dummy) {
2402     return 1;
2403 }
2404 }
2405 static Uint exec_seq( PTrafo *args) {
2406     stackpush( &(amp; peu->pstack), args[ 1]);
2407     stackpush( &(amp; peu->pstack), args[ 0]);
2408     return 1;
2409 }
2410 static Uint exec_equ( PTrafo *args) {
2411     PNode p;
2412     Uint t = 0;
2413     p = peu->pnode->pfirstsub;
2414     if ( ( p != NULL) AND cmp_value( p->value, peu->pnode->value, &t)) {
2415         stackpush( &(amp; peu->pstack), args[ 0]);
2416         if ( traceflag)
2417             printf( "+");
2418     }
2419     else {
2420         stackpush( &(amp; peu->pstack), args[ 1]);
2421         if ( traceflag)
2422             printf( "-");
2423     }
2424     return t+1;
2425 }
2426 static Uint exec_all( PTrafo *arg) {
2427     static ExecUnit tmpeu;
2428     Uint nofchildren = 0;
2429     PNode p;
2430     if ( ( p = peu->pnode->pfirstsub) != NULL) {
2431         do {

```

```

2440         tmpeu.pnode = p;
2441         tmpeu.pstack = NULL;
2442         stackpush( &tmpeu.pstack, *arg);
2443         tmpeu.parent = peu;
2444         execcreate( &tmpeu);
2445         nofchildren++;
2446         p = p->pnext;
2447     } while ( p != peu->pnode->pfirstsub);
2448     peu->childcnt = nofchildren;
2449     peu = execuget();
2450 }
2451 return nofchildren;
2452 }
2453 static Uint exec_trace( PTrafo *dummy) {
2454     traceflag = NOT traceflag;
2455     if ( traceflag)
2456         printf( "TRACE");
2457     return 0;
2458 }
2459 static Uint exec_trace0( PTrafo *dummy) {
2460     if ( traceflag)
2461         puts( "");
2462     traceflag = NO;
2463     return 0;
2464 }
2465 static Uint exec_trace1( PTrafo *dummy) {
2466     traceflag = YES;
2467     printf( "TRACE");
2468     return 0;
2469 }
2470 static Uint exec_dump( PTrafo *dummy) {
2471     if ( NOT traceflag) {
2472         nodeprint( peu->pnode, stdout, NO);
2473         puts( "");
2474     }
2475     return 0;
2476 }
2477 static Uint exec( void) {
2478     PTrafo pacttrafo;
2479     Uint time, tx;
2480     if ( traceflag) {
2481         printf( "EXEC\t - ");
2482         nodeprint( peu->pnode, stdout, NO);
2483         puts( "");
2484     }
2485     time = 0;
2486     while ( peu->pstack != NULL) {
2487         pacttrafo = stackpop( &( peu->pstack));
2488         switch ( pacttrafo->type) {
2489             case TT_SET:
2490                 if ( traceflag) {
2491                     printf( "set ");
2492                     valueprint( pacttrafo->select.value, stdout);
2493                 }
2494                 discard_value( peu->pnode->value);
2495                 peu->pnode->value = copy_value( pacttrafo->select.value, &tx);
2496                 ++tx;
2497                 if ( traceflag) {
2498                     printf( "\t - ");
2499                     nodeprint( peu->pnode, stdout, NO);
2500                     puts( "");
2501                 }
2502                 break;
2503             case TT_BIN: {
2504                 BinCode bc;

```



```

2515     if ( ( bc = pacttrafo->select.bin.bc) > NOF_BUILTINS) {
2516         printf( "unknown trafo code\n");
2517         abort();
2518     }
2519     if ( traceflag AND ( bc != BC_seq)) {
2520         printf( "%s", trafotable_search( builtin[ bc].namecode));
2521     }
2522     tx = execfunc[ bc]( pacttrafo->select.bin.args);
2523     if ( traceflag && ( bc != BC_seq)) {
2524         printf( "\t - ");
2525         nodeprint( peu->pnode, stdout, NO);
2526         puts( "");
2527     }
2528     break;
2529 }
2530 case TT_LOOP:
2531     if ( traceflag) {
2532         printf( " - LOOP -\n");
2533     }
2534     stackpush( &( peu->pstack), pacttrafo->select.loop);
2535     tx = 0;
2536     break;
2537 default:
2538     printf( "internal error: invalid trafo type\n");
2539     abort();
2540 }
2541 time += tx;
2542 }
2543 return time;
2544 }
2545 }
2546 static Uint exectime;
2547
2548 void executor( void) {
2549     traceflag = NO;
2550     exectime = 0;
2551     while ( ( peu = execuget()) != NULL) {
2552         exectime += exec();
2553         if ( ( peu->parent != NULL) AND ( --peu->parent->childcnt == 0))
2554             execurevive( peu->parent);
2555         execufree( peu);
2556     }
2557 }
2558
2559 void execute() {
2560     ExecUnit start;
2561
2562     execfunc[ BC_exch] = exec_exch;
2563     execfunc[ BC_cdn] = exec_cdn;
2564     execfunc[ BC_cup] = exec_cup;
2565     execfunc[ BC_insert] = exec_insert;
2566     execfunc[ BC_delete] = exec_delete;
2567     execfunc[ BC_left] = exec_left;
2568     execfunc[ BC_right] = exec_right;
2569     execfunc[ BC_split] = exec_split;
2570     execfunc[ BC_replace] = exec_replace;
2571     execfunc[ BC_skip] = exec_skip;
2572     execfunc[ BC_seq] = exec_seq;
2573     execfunc[ BC_eql] = exec_eql;
2574     execfunc[ BC_all] = exec_all;
2575     execfunc[ BC_trace] = exec_trace;
2576     execfunc[ BC_trace0] = exec_trace0;
2577     execfunc[ BC_trace1] = exec_trace1;
2578     execfunc[ BC_dump] = exec_dump;
2579
2580     nodememinit( 10000);
2581     stackinit( 10000);
2582     execuinit( 1000);
2583
2584     start.pnode = nodeallocfill( setatom( 0), NULL);;
2585     start.pstack = NULL;
2586     stackpush( &start.pstack, program);
2587     start.parent = NULL;
2588
2589     execucreate( &start);

```

```

2590
2591 //printf( "\ninput: ");
2592 //nodeprint( start.pnode);
2593 //puts( "");
2594 puts( "executing...\n");
2595
2596 executor();
2597
2598 printf( "\nexected, time = %d.\n", exectime);
2599 //printf( "\noutput: ");
2600 nodeprint( start.pnode, stdout, NO);
2601 puts( "");
2602 }

```