

# Concurrent Object-Oriented Programming in Lisp

J. PADGET, R. BRADFORD AND J. FITCH

*School of Mathematical Sciences, University of Bath, Claverton Down, Bath, BA2 7AY*

*We describe the approach to the concurrent execution of object-oriented programs that is being researched at the University of Bath. The aim of this project is the concurrent execution of both new Lisp programs and existing Lisp applications. We are pursuing this goal by addressing the problems of concurrent execution at several levels: static analysis of Lisp programs as a basis for semi-automatic transformation, modification of medium-sized Lisp programs to use concurrent primitives to gain experience in their use and behaviour and the implementation of various concurrency primitives in a compiled distributed heterogeneous Lisp environment. Each of these topics is described in turn and its relationship to the long term aim defined above is examined.*

*Received January 1991*

## 1. INTRODUCTION

The aim of our project is to take new and existing Lisp applications and execute them on closely coupled multi-processors or distributed heterogeneous processors. This paper describes our programme for achieving this and reports on the progress we have made towards this goal. Consequently, a proportion of this paper discusses what we have built and the rest discusses what we intend to build. Our plan is to use a semi-automatic compilation system which will convert the sequential program into one using the concurrency primitives we have defined. Clearly, this is a difficult problem and one which we have not yet solved. We believe we can arrive at a solution via a constructive approach in three stages:

(1) Developing the concurrency primitives. These are our foundation. The concurrency primitives currently in use are *futures*,<sup>19</sup> *linda*<sup>7</sup> and *time-warps*.<sup>22</sup> These operations have all been implemented in a compiled Lisp environment which executes on a heterogeneous network. Modules implementing these operations have also been written for EuLISP<sup>32</sup> and executed on both tightly- and loosely-coupled multi-processing environments.

(2) Writing or rewriting programs with explicit concurrency constructs. This helps us to test the reliability and generality of the mechanisms. At the moment we have a version of OPS5<sup>12</sup> rewritten in an object-oriented style, in which the RETE network is represented as a collection of objects.<sup>2</sup> Another major Lisp program we use is the Reduce algebra system.<sup>20</sup> Some of the algorithms in Reduce, in particular those for Gröbner bases and for Hensel lifting, are particularly suitable for concurrent execution. We intend to rewrite these parts using explicit concurrency constructs. The Gröbner basis code in Reduce has been modified for concurrent execution using Linda and run successfully. The third application area is discrete event simulation and we will be adapting the ARSONIST simulation (a forest-fire fighting simulation) for concurrent execution. Since ARSONIST is already an object-oriented program, this will give an opportunity for more abstract experimentation since the parallelism can be written into the metaclasses rather than appearing directly in the program.

(3) Static analysis of programs to gather information for the semi-automatic compilation from sequential to parallel programs. A preliminary data-flow analyser has

been written and applied to the factoriser of the Reduce system,<sup>9</sup> but the results of this analysis have yet to be put to use. A second, more advanced static analyser is being developed.<sup>23</sup>

At the end of these stages we hope to have collected sufficient knowledge about the behaviour of concurrent programs to enable us to build a general-purpose concurrent execution environment.

An orthogonal issue to concurrent execution is that of object management, or, more bluntly, the size of the object-oriented program. This is a particular problem for both OOPS5 (the object-oriented version of OPS5) and for discrete event simulation. In the case of OOPS5, the largest rule-set we have run generates a network containing 5,500 objects. In the case of ARSONIST, we do not have any such large figures, because, at present the simulation is relatively simple. However even a realistic modelling of the terrain would need at least 500 objects per square kilometre. The planning capabilities of the fire-fighters can be improved to a certain extent by algorithmic means, but in the longer term, we intend to provide a separate expert system for each planning agent in the domain. The problem is one of scale: quite small simulations are certainly going to consume all available physical memory, it is even conceivable that large simulations could exhaust a 32-bit address space. Our solution to this is persistence and object caching.

At present, we are working in two Lisp environments: one based on Portable Standard Lisp (PSL),<sup>16</sup> which provides a compiled heterogeneous distributed Lisp environment supporting *futures*, *linda* and *time-warp* and the other based on EuLISP, which provides an interpreted (byte code and native code compilation are under development) environment with the same facilities. OPS5 has been rewritten in an object-oriented style – as a first step towards conversion for concurrent execution – and behaves equivalent to the original program. Some parts of Reduce have been analysed, the Gröbner basis part has been modified and a re-write of power-series has been planned. There is a dataflow analyser which has been applied to some non-trivial programs.

## 2. CONCURRENCY PRIMITIVES

Identifying concurrency in a program is one problem; expressing that concurrency effectively is another. The

degree of abstraction provided by the concurrency operations is important because that affects how readily concurrency can be expressed. We believe semaphores, critical regions and even occam style processes are too low-level to be of use in the source language. Therefore we seek something with more abstraction. At present we have chosen to use three concurrency abstractions. Two are quite similar: *futures* and *linda*. The third – *time-warp* – is somewhat different and at a higher level.

Why have three ways of expressing concurrency? Our answer is that there are different granularities of concurrency in a program and different abstractions capture different granularities. Although, the system, of necessity, has primitive concurrency operations, these are there to construct concurrency abstractions. In their turn, these abstractions incorporate declarative information about the nature of the concurrent process. In our opinion, based on empirical observations, *time-warp* is suitable for very coarse-grain concurrency and *futures* and *linda* are suitable for medium-grain concurrency. The latter two have complementary features which makes each attractive depending on the circumstances: *linda* inherently offers a means of limiting parallelism, whilst *futures* is simply eager evaluation with no self-imposed limits. The attraction of *time-warp* lies in its *speculative* evaluation tactic. However, its effectiveness is yet to be fully proven.

Rather than pursue one method of expressing concurrency, we decided that we wanted to get practical experience of several. That is to say, we are quite prepared to add other concurrency models to our system in order to learn more (for example, a CSP model). Consequently, we have developed an implementation of Lisp which runs on different kinds of processors on a local area network to provide us with a cheap distributed processing environment, work which was carried out partially in cooperation with the RAND Corporation. Within this Lisp we have implemented multiple control threads and used that to implement *futures*, *linda* and *time-warp*. This allows programs to be written that use any combination of these concurrency primitives. More recently, we have developed a new implementation of Lisp following the EuLISP definition. This is a more attractive long-term development vehicle because support for parallelism has been built into the design of EuLISP and the abstraction facilities provided by its module mechanism and fully integrated object system (TEAOΣ) are what we need for both robust programming and fast prototyping.

## 2.1 Light-weight processes

The PSL system was developed without the benefit of a multi-processor, so the multiple-thread facility is somewhat limited. Since then, the arrival of EuLISP and of two Stardent Titan-3 (each with three processors) have led to the development of true parallel executing threads in our implementation of EuLISP.

The EuLISP thread model provides some primitive operations for the creation and manipulation of threads, but intentionally avoids attempting to be high-level. Thus the basic operations are: *make-thread* to create a thread, *proceed* to indicate that a thread can continue executing and *suspend* for a thread to give up executing voluntarily. The goal for the EuLISP thread primitives is

to provide sufficient power to implement more abstract and more usable concurrent processing models without being biased towards any particular one and without being biased to a particular architecture. Current evidence, based on our implementations of *futures*, *linda* and *time-warp* on top of these primitives, leads us to believe that they are satisfactory. Clearly, some new abstract parallelism model may yet modify this situation.

Process abstraction is another area in which objects play a key role in this project. Although a number of the applications we are using are object oriented already, that does not necessarily make them executable concurrently. To make the transition from serial OOP to concurrent OOP, we will use TEAOΣ<sup>15</sup> to define new metaclasses which incorporate concurrent execution, so that applications written in terms of those metaclasses can execute concurrently without major changes to the source code of the application. EuLISP is different from most other Lisps in that the object system is tightly integrated with the Lisp – indeed the basic Lisp types are classes – and so threads are already part of the class hierarchy. Indeed the implementations of *futures* and *linda* in EuLISP are simply classes built on top of the thread class. In order to simplify the scheduling of these different types of processes which can exist simultaneously, the notion of the scheduler as a generic function – which dispatches on the class of the process it is being asked to schedule – is being developed.

## 2.2 Distributed execution

The concurrency primitives are independent of any particular machine architecture or topology. At present we are using them in a distributed environment built from different Unix workstations linked by ethernet. This provides us with a cheap, loosely-coupled, private-memory multi-processor. Clearly, such an environment is not satisfactory in the long term, but it has worked well as a test-bed on which to develop the primitive operations.

The distributed execution environment has been built by adding network operations to the Lisp system and a means to start Lisp images on several machines. All the Lisp processes then try to establish network connections with all of the other Lisp processes. The result is a fully connected virtual network of Lisp processes which read and write over sockets.

This environment has forced us to face issues peculiar to private memory systems. In particular, a great deal of effort has been devoted to the network transmission model<sup>4</sup> and into process migration techniques. Recent work at RAND on the latter has been very effective, allowing *time-warp* processes to migrate during the execution of the program, resulting in better processor utilisation and much faster run-times.

## 2.3 Futures

The *future* concept is well-documented.<sup>19, 25, 29</sup> The idea can be summarised briefly by considering the evaluation of (*future expression*). Unlike an ordinary function call, *expression* is not evaluated and then passed to *future*, instead a new process is created to manage its evaluation. The call to *future* does not wait for the process evaluating *expression* to be completed, instead it returns a *future* object, which is a handle on the *expression*

process. The process which called *future* now proceeds concurrently with the evaluation of *expression*. The *future* object will contain the result of *expression* when that process terminates, but if any process tries to access the value of the *future* before the *expression* process is completed, it is blocked.

A disadvantage of *future* at the implementation level is that the application of certain operators must check for *futures* and treat them specially. The consequence is a processing overhead to check whether such an operator has been given a *future* or not. In brief, anywhere there is a strict function or a function that is strict in some particular argument, it must be checked for a *future*. In ref. 25 it is reported that this overhead slows the system down by a factor of two. An alternative tactic to handle *futures* is to make every function that might dereference a *future* into a generic function and then write corresponding methods. This would have the effect of moving the testing cost into the generic function dispatch mechanism. Clearly, it would not reduce the cost in any way, just move it elsewhere. A recently developed technique<sup>34</sup> suggests a lower overhead implementation is possible.

The *future* is a very simple and attractive concurrency abstraction. It is also simple to use: the program is examined to decide where a significant amount of work could be done in parallel and then that function call is wrapped with *future*. This is the positive side.

As with any powerful operation, it is quite easy also to make mistakes. Mistakes in this context are: too many *futures* and wasted *futures*. The first clogs the system and makes it *thrash*. The second slows down a concurrent program by making tasks insufficiently complex, so that the cost of process management becomes more significant than the time saved by executing concurrently. This is why deciding where to place *futures* in a program is a subtle task. Qlisp<sup>13</sup> puts dynamic control over the creation of *futures* in the hands of the programmer – the drawback is that such control is local, whilst the problem is of a global nature. Recent work on conditional task creation<sup>26,30</sup> recognises this fact. Implementations of *futures* have been done for PSL (uniprocessor) and for EuLISP (uni + multiprocessor).

## 2.4 Linda

The *linda* process model<sup>7</sup> is not so different from the *future*. Whilst a *future* is created for each expression that is the argument of *future*, the *linda* user creates a pre-determined number of process for each kind of expression in advance. In effect, *linda* is a software form of dataflow.

The *linda* model capitalizes on the probability that the majority of operations that could be done concurrently are of the same class, but with different parameters. If we consider *quicksort*, one might insert parallelism by processing each of the partitions in parallel. Using *futures*, this would suggest wrapping each recursive call, but the *future* operation would always be the same: the *quicksort* function. The difference is the parameter, which is a partition of the input list. Of course, this is a very particular example and *quicksort* is not a very representative program. However, we have observed that often the operation to be performed by many of the *future* processes is the same; only the arguments differ. In Linda, we can create as many servers for a particular

operation as appropriate (a decision to be based on resources and the relative importance of this component of the parallel computation in question) rather than an arbitrary number directly proportional to the magnitude of the input.

In brief, the *linda* model consists of a collection of processes. In that collection there can be many sets of identical processes. The number of identical processes of a given type offers a means of controlling the amount of concurrency of a particular kind of operation. We will return to this property of *linda* later. Processes in the *linda* model communicate through special operations on a pool of data. The originators of *linda* call this pool the *tuple-space* – more recent developments include the use of multiple pools organised in hierarchies. The *tuple-space* contains data being communicated from one process to another. However, to make sure that the right kind of process picks up the right kind of expression, the expressions are tagged. A process takes an expression from the pool using the operation *in* and puts an expression into the pool using the operation *out*. A process takes a copy of an expression from the pool with *read*. The *eval* operation puts an unevaluated expression into the pool and creates a process to evaluate it – this is called an *active tuple*. When the evaluation is complete, the tuple becomes *passive* and can be taken from the pool using either *in* or *read*. Clearly, *eval* is very similar to a *future* operation and, indeed, could be implemented as such. Thus, we start to see the benefits of having several concurrent paradigms in the same environment.

The operation of taking an expression from the pool can be quite complex and in the full *linda* model involves pattern matching. Patterns can be seen as a logical extension of the use of tags to identify for which process a tuple is destined.

Although *linda* is, like *futures*, an abstract model, independent of architecture, the tuple-space communication model is inherently well suited to a shared-memory multi-processor. If they had not existed, all the networking operations would have had to be implemented to support *linda* in a distributed environment. In practice, we have found managing the tuple-space in a distributed environment to be fairly straightforward since we were able to build on the communications code already installed – and well-debugged. Implementations of *linda* have been done in PSL (uniprocessor and distributed multiprocessor) and in EuLISP (uni + multiprocessor, a distributed version is in progress).

## 2.5 Time-warps

The *time-warp* process model is somewhat different from *futures* and *linda*. We call the *time-warp* model a *speculative* evaluation model. The reason is that, under *time-warp*, computation is often done before it is required to be done. Indeed, it is often not known whether the result of a computation is even necessary. The *time-warp* model gambles on what computations will be needed in the future. So far, *time-warp* sounds interesting, if somewhat wasteful. The negative side of *time-warp* shows up when it is discovered that the results of a computation undertaken speculatively are not needed. In a purely functional world this would not be a problem.

In a side-effect world, the side-effects have to be undone. In *time-warp* terminology, this operation is called *rollback*. It is the *rollback* issue that determines most peoples' attitude to *time-warp*; some consider it completely unreasonable and reject the whole *time-warp* approach, others consider it as a serious, but not insurmountable, problem. The dilemma is captured in the *time-warp* thesis,<sup>22</sup> which is:

- that *rollback* occurs only infrequently (temporal locality);
- that the cost of *rollback* is no more than what would have been wasted by not doing any speculative computation;
- that there is a simple implementation of *rollback*.

The issue is whether this thesis is credible or not. We have decided to test it in an implementation. We have no definitive answer yet, but we still feel positive about the *time-warp* approach. We have been involved with an initial implementation of *time-warp* at the RAND Corporation, and two separate implementations at Bath, and so can say that in practice, the *time-warp* model fits in very well with object-oriented evaluation and, indeed, with concurrent evaluation.

In brief, the *time-warp* model is of a collection of processes where process interaction is carried out by asynchronous message passing. The core of the technique lies in how the messages are handled, but in order to describe how *time-warp* operates, first the concept of *virtual time* needs to be explained.

Virtual time in a *time-warp* program starts at zero and progresses to infinity. Each *time-warp* process has a local virtual clock which records the *local virtual time* (LVT). Virtual time is propagated through the network of processes by the operation of message passing. When the LVT of each process in a program reaches infinity, this indicates the termination of the whole *time-warp* program. Thus, virtual time measures the progress made by a *time-warp* program towards the end of its computation.

Each message sent is stamped with the LVT of the sending process and an estimated receive time which is strictly greater than the LVT of the sending process. Each incoming message is queued until the *time-warp* process is ready to evaluate it. The messages are stored in receive time order and the LVT of the receiving process is advanced to the receive time stored in the message immediately before evaluating the message. Thus, virtual time is propagated through the collection of processes.

Rollback occurs when a message arrives late. That is to say, the receive time on the message is less than the LVT of the receiving process. The receiving process must now restore the state that existed at the receive time of the message and start computing again from that time. The restoration of the state of an object can be done by taking snapshots of the slot values of the object between processing each message, say. Unfortunately, restoring the object's internal state is only part of the picture. Between the virtual time at which the message should have been processed and the current LVT of the object, messages might have been sent to other *time-warp* processes. Hence, those messages must be recalled and, perhaps, those processes rolled back too.

Undoing the effects of messages is achieved by using

the so-called *anti-messages*. For each message that must be recalled, a corresponding anti-message is sent. If the message and the anti-message meet in the *time-warp* object input queue, they cancel each other out. If the message is not in the input queue, then it might not yet have arrived, or it might have been processed already. In the former case, the anti-message is inserted to await the arrival of the message, and they will then annihilate. In the latter case the process accepts the anti-message immediately and starts to roll back to the time of the anti-message. This might well cascade to many more objects. However, since the algebraic sum of messages in the system – that is messages and (real or potential) anti-messages – is zero and because of the requirement that the receive time of a message be strictly greater than the send time, the rollback operation is guaranteed to terminate.

What has been described here is the principles involved in *time-warp* operation. The overheads make it seem impossibly expensive. However, like many algorithms, the simple explanation and the naïve implementation are good for understanding how it works, but unreasonable in practice. There are many differences between the description here and an efficient implementation.<sup>14</sup> Implementations of *time-warp* have been done for PSL (constructed on top of *linda* on a uniprocessor) and for EuLISP (distributed multiprocessor).

## 2.6 Process migration

The use of a distributed system has encouraged us to investigate the question of process migration. Clearly, it is much harder to change the processor running a process if the system is distributed than if it has shared memory. In fact, a shared memory processor makes it possible to ignore process placement and process migration issues for much longer than with a distributed system. It became apparent to us early in our research that although initial process placement was not unhelpful, some dynamic technique was needed.

Migration of *time-warp* processes has been implemented on the RAND system and has proven very successful in the sense that the *time-warp* test programs ran about an order of magnitude faster with process migration than they did without. The details of the migration policies and their relative effectiveness are to be found in ref. 6. More recently, we have developed an idea, which is currently being implemented at Bath, that uses *futures* as a means of controlling migration (described below).

## 2.7 Migrating time-warp processes

A *time-warp* process is represented as an object. The object comprises some slot values, the state queue (used in rollback), the input message queue and the output message queue. We impose the restriction that a *time-warp* process may only migrate in between message processing cycles. In this way we are assured that the state of the *time-warp* process is consistent when it is migrated. Consequently, the *time-warp* object is simply a data-structure to be transferred from one memory to another. This can be accomplished using the network primitives mentioned earlier.

## 2.8 Futures as a migration medium

The migration of *time-warp* processes described above could be called *stop-and-copy* because the process is halted, copied and restarted. This is probably an appropriate strategy for a data structure such as a process, which is likely to need to make frequent reference to most of its state. In the general case of migration of structure between processors, this might not be true. This concern has led researchers to talk of *copy-on-reference* and *lazy-copying* as means of avoiding copying large data structures of which only small parts might be referenced.

Using *futures* as a control abstraction, an elegant model for data structure migration has been developed. This model can support copy-on-reference, lazy-copying and even move-on-reference and lazy-moving. The technique is currently being implemented at Bath<sup>21</sup> and a similar approach<sup>33</sup> is being used in the ICS-LA (Implementation Compilation et Sémantique des Langages Applicative) project at INRIA. In brief, the method works by creating a *future* for each remote reference. Then, the process related to the *future* can either copy the data-structure when a process blocks on access to the *future* or copy it as a background process interleaved with other computations. There are two interesting related issues:

1. limiting the amount that is copied;
2. maintaining the integrity of the value to which there is a remote reference.

Limiting the amount of structure copied is an issue for copy-on-reference. Two options immediately apparent are depth-first and breadth-first copies – which is preferable will, in all likelihood, depend on the application. In the breadth-first case, for instance, copy-on-reference would traverse the structure making a new *future* object for each place where the structure might be descended further. Further accesses to the structure might encounter other *futures* and thus cause more copying. Hence the actual amount of structure moved can be limited to what is accessed, but at the price of waiting for those parts on almost every access.

The second issue is maintaining integrity of a value after a reference to it has been exported. The remote process must be able to refer to the value at the time of exporting the reference. Therefore, it must be protected against any changes to the value made by the process in whose address space it resides. In fact, rather than being protected against changes, it is necessary to record the changes so that previous values can be recovered. Such behaviour is very similar to that provided by a *time-warp* process. The significant difference between the needs of this object and a true *time-warp* object is that there is no need to support rollback since old values can be determined simply from examining the saved states of the object. Thus, we again see an advantage from having several concurrent paradigms in the same environment.

An extension of this idea using *futures* to copy data structures can also be used to distribute a data structure across processors. In this way the issue of maintaining integrity can be avoided, because the uniqueness of the structure is preserved. The operation of handling a request for a remote reference is much as described

previously. The difference is that instead of making an object to handle the updates to the value, a remote reference is installed in its place and the processor making the remote reference now becomes the owner of the value. Thus, a data structure can be moved rather than copied between processors and hence a data structure may be distributed across several processors. Such a technique might be attractive for divide-and-conquer style algorithms in which different processes worked on different parts of a common data structure.

## 2.9 Persistent objects

The final part of our infrastructure for supporting large scale concurrent applications is a mechanism for persistent objects. In the introduction, we outlined our need for persistence: the problems we want to execute are very large. Persistence will have two benefits: first is that we can run very large problems, even to the extent of problems that could not fit in a 32-bit address space, second is that it will also aid efficiency in a manner analogous to generational and ephemeral garbage collection by keeping the working set size down.

The persistent object system in EULISP is a result of porting the Persistent Simulation Environment<sup>3</sup> from Common Lisp, which in turn is an adaptation of the Picasso system<sup>35</sup> developed at Berkeley. However, PSE is a simplification of Picasso, in that it uses flat files instead of Postgres (the successor to Ingres.) Because the persistent system is built on top of TELOS it has the twin advantages of integrating the resulting system with the rest of our development environment and making development of the persistent facilities simpler since it is only an extension of the existing architecture. In brief, the approach has been to define a new slot description class for persistent slots (in objects) which implements the object cache and pre-fetch policies. The accessing of a slot in an object is mediated by the slot-reader, which is a generic function on the slot description class, hence, the slot access can load the object, if necessary, and return the contents of the slot. In effect, this looks much like virtual memory management. As with virtual memory, where it is desirable to try and ensure that the page is loaded before each reference, we would like to ensure that each object is loaded before the message is sent. For this, we need pre-fetch policies and, indeed, replacement policies. In the case of the discrete event simulations, where a vast number of the objects are map and feature data – which rarely change – a pre-fetch policy is quite straightforward since there is a strong likelihood that adjacent map segments and related feature data will be needed. In the production rule system too, there is locality in terms of the nodes in the RETE network and therein the basis for different pre-fetch policy which will suit this problem.

The persistent object system has been used to run some small simulations based on US Army map data to find shortest paths between locations and to simulate an activity network. Under development at the moment is a Petri net simulation language using persistence. Complementary to this work, an interface has also been developed to the Unix database manager, dbm (in fact, to the GNU version, called gdbm, which supports access to multiple databases). Again, some small experiments have

been run on sample map data, this time supplied by Bartholomews, covering a 100 km<sup>2</sup> region centred on Bristol.

### 3. MANUAL PARALLELISATION

Implementing a concurrent operation such as *future*, *linda* or *time-warp* is only the first step towards concurrent processing. What is needed more than anything else is experience in the use and effectiveness of these concurrent operations. Initially, one needs to know whether the implementation is correct – as far as can be told from empirical observation – and then one wants to find out how easy it is to use the concurrent operation in practice. In the long term we expect the concurrent operations to be inserted semi-automatically into the application program. In the short term it is part of our programme to modify existing Lisp applications in order to gain practical experience. Not least, this experience will be useful in developing the semi-automatic paralleliser.

We have taken two quite well-known small- and medium-sized Lisp programs as the basis of our experiments in manual parallelisation: the OPS5 production rule interpreter and the Reduce algebra system. We are working on each of these in quite different ways. OPS5 has been rewritten as OOPS5. Reduce is being analysed – as a prelude to rewriting and to test the analyser – and one part of it has been rewritten (many other parts remain). The third program, called ARSONIST, was developed at RAND for forest fire simulation.

#### 3.1 OPS5

The OPS5 production rule interpreter is, in its original form, a small program written in Franz Lisp. It has been ported to many different Lisps including Common Lisp, Cambridge Lisp, PSL and Le-Lisp. The interpreter comes in two parts: one builds the so-called RETE network from the left hand sides of the productions, the other drives the recognise-act cycle of the interpreter. Briefly, a production rule interpreter works by matching the current state of the working memory against the left hand sides of the production rules to generate the *conflict set*. This is the recognise phase. The conflict set is the set of productions which match the current state of working memory. The act phase selects one production from the conflict set and takes the actions specified on the right hand side of that rule. Then, a new recognise phase starts. As with most other work on parallelising production system<sup>17,18</sup>, we are concentrating on the recognise phase of the recognise-act cycle.

The interpreter has been rewritten to use six different classes. For brevity here, we assume familiarity with OPS5 terminology, but for details see ref. 12. The six classes are:

1. production rule;
2. working memory distributor;
3. condition element;
4. conflict resolution manager;
5. working memory element;
6. working memory clock.

The first part of the interpreter has been rewritten to

use objects so that the RETE network is represented by a collection of instances of condition elements and working memory elements. Testing on several widely used rule-sets has shown the rewritten OPS5 to behave in an equivalent manner to the original OPS5 program. This is the current state of the project. Having objectified the network we can now consider executing it concurrently on a distributed network. Each of the condition elements and working memory elements can now be treated as a separate process and distributed across the multi-processing system. Consequently, searches through the network can be executed concurrently. However, although it is widely recognised that production systems spend the majority of execution time in the recognise phase, there is still a bottleneck on parallelism in the act phase. The act phase can be likened to the commit operation in a database – having selected the rule to fire, we must await the completion of the execution of the right hand side before starting the next recognise iteration.

The use of *time-warp* style execution offers an interesting way of avoiding the synchronisation at act time. Because a *time-warp* object is able to rollback to previous states – it could be likened to backtracking – if the working memory elements could record different values corresponding to different virtual times, a similar effect could be achieved. Thus, the recognise-act cycles can be overlapped and the act phase does not require a synchronisation once for every loop of the interpreter. The support for this has been implemented but it has not been tested seriously yet. This work has been carried out in PSL using a uniprocessor.

#### 3.2 Reduce

We are using Reduce in two ways. As a source of large amounts of Lisp to feed to the dataflow analyser (see section on Semi-Automatic Parallelisation) and as a test-bed for manual insertion of concurrent constructs into programs. These two uses provide mutual feedback, since the dataflow analysis suggests where there is parallelism to be released and the manual insertion (and inspection) suggests where the analyser has not discovered enough information. However, there is another reason for the interest in computer algebra. The reason is that computer algebra problems consume large amounts of time (and space) and a number of the common algebraic algorithms offer a lot of potential for parallelism.

The two algorithms of primary interest are the Hensel lifting stage of polynomial factorisation and Gröbner basis computations.<sup>28</sup> Gröbner bases are becoming the lynch-pin of many new algorithms proposed in the computer algebra community as well as being fundamental in solving algebraic problems in robotics related to the movement of objects in confined spaces. Hence, computing Gröbner bases quickly is becoming increasingly important – and increasingly difficult for a uni-processor as the polynomials get larger and more numerous. A third area is power series evaluation – since the Reduce implementation of power series<sup>31</sup> is modelled as a network of streams, this could easily be transformed into a network of communicating processes.



### 3.3 Discrete event simulation

At present we only have one discrete event simulation program; the ARSONIST forest fire fighting simulation. In the case of simulation, it does not matter so much whether the program is widely known or used – as distinct from our other choices – because it is the mechanisms of discrete event simulation that are important and much less so the scenario being simulated. However, the size and sophistication of the simulation do affect the generality of the results.

The programming language for the simulation is a development of the RLISP language in which Reduce is written, called RLISP88. RLISP88 is an object-oriented language designed to support the writing of discrete event simulations and is implemented as a parser from RLISP88 to Lisp running on top of Lisp.

The ARSONIST simulation comprises a hypothetical map, the grid positions of which are either trees, grass, dirt, houses, water, ash, fire-break or fire. The fire-fighters are bulldozers which roam across this terrain building fire-breaks to contain the fire with the primary goal of saving houses and the secondary one of saving trees. There is a single control centre which has aerial reconnaissance to track the spread of fires and radio contact with each of the bulldozers. Scenarios are created by starting fires in various places and specifying factors such as wind speed and bearing and the ease with which materials will catch fire. By changing the root class of the objects in this system and changing how messages are passed, we will be able to experiment with concurrent execution.

## 4. SEMI-AUTOMATIC PARALLELISATION

To develop our intuitions and then our knowledge of the behaviour of concurrent programs, we believe we have to start by (re)writing programs with explicit concurrent operations. However, as stated at the beginning, our long term goal is an environment for the development of concurrent programs where the concurrent operations are inserted automatically or semi-automatically. To address this problem one dataflow analyser has been developed and another, more advanced, is under development. The purpose of these programs is to make a static analysis of a program written without concurrent constructs and use the information to insert or to suggest where to insert concurrent operations. This is one part of semi-automatic parallelisation. This is a means to identify what can be done concurrently. The second part of semi-automatic parallelisation is to decide whether it is worth executing something concurrently – static estimation of run-time. The third part is to modify what is done concurrently based on observed behaviour – dynamic analysis.

We are working with two dataflow analysers: one is based on the ideas in ref. 27 developed in ref. 10 and taken further in ref. 9. This only produces a record of the analysis. As yet, there is no integration with any compiler. The other analyser is being developed<sup>23</sup> and has produced analyses and rewrites of some quite complex test cases.

### 4.1 Static analysis

The current dataflow analyser takes functions, or other fragments of code, and constructs a flow graph from it. The analyser contains tables of semantic information about the basic Lisp functions and rules about composing this semantic information depending on the form of the program being analysed. This flow graph is processed using this semantic information to yield a semantic description of the side effects of each function of code fragment. In effect this description is an annotated closure of the function identifying the non-local effects of the function.

The description is a 4-tuple:

1. **read only** non-local references;
2. **read/write** non-local references and modifications, of which the latter implies a need for exclusive access;
3. **write only** non-locals modified (before reference, or not referenced at all);
4. **hard** a boolean which is true if nothing could be determined about the side-effects of the function (expressions involving `set` or `nonc` might cause this).

The resulting descriptions may be saved in a file and input to the analyser to provide a means of adding to the analyser's knowledge of the program on which it is working. Hence one can analyse a module at a time but provide the semantic information about each module as needed when working on a large program. An issue that is often overlooked in parallelism is whether code that can be executed in parallel is worth executing in parallel – that is, the grain size is too small for the overheads of a given architecture. We measure the complexity of a piece of code using a technique called the static estimation of run-time, which is explained in the next section.

The new static analyser<sup>24</sup> uses new techniques for representing and analysing the program based on the ideas in single static assignment (SSA)<sup>8</sup> and the PTRAN analyser.<sup>1</sup> The new analyser also builds on ideas from the current analyser, in particular with respect to the cost of executing a block concurrently. The task is seen as one of partitioning the program for concurrent execution. Although this problem is NP-complete, a heuristic approximation algorithm has been developed with near optimal behaviour. The algorithm works by taking the units of finest grain parallelism in the program and composing them to build successively coarser grains of parallelism until the units are sufficiently large to cross the cost function (communications and scheduling overhead) of the architecture for which the program is being prepared. Effectively, the partitioning algorithm tries to minimise the cost function for a given architecture. Some preliminary results of this algorithm are in preparation.

### 4.2 Static estimation of run-time

Discovering what can be executed concurrently is a hard problem. Deciding whether it is worth executing concurrently is insoluble. However, one can make a good guess.<sup>11</sup> We call it a *good* guess because we have been surprised at the accuracy of the guesses made by the run-

time estimator. The static estimation of run-time concerns trying to guess how complex is a given piece of program and therefore whether the overheads of managing its concurrent execution are greater than the benefits of its concurrent execution.

Determining the complexity of a program by static analysis is equivalent to determining whether a program terminates. Hence, it is insoluble. But to avoid creating tasks with too fine a grain of parallelism some estimate of what is and is not worth executing concurrently can be very useful. The practical results of estimating the run-time of a program have been surprisingly, even frighteningly, accurate. The approach is constructive, as might be expected, in that each of the primitive operations is assigned a cost relative to the cost of a particular primitive. In this case, the base primitive chosen is *car*. Each basic operation is timed on a given processor to obtain the spectral analysis of the Lisp operations on that processor. These data are then used in the last stage of the dataflow analysis to estimate the cost of each identified concurrent unit and, hence, provide a basis for a decision about whether to execute that unit concurrently.

#### 4.3 Dynamic analysis

In the first instance a static analysis can determine the major independent regions of the code and a static estimation of run-time can help make decisions about whether it is worth separating off a task. However, the dynamic behaviour of a concurrent program is difficult to predict and almost as hard to observe. A multi-processor profiling tool has been developed at RAND as a first step in this direction<sup>5</sup> and we hope to be able to use the trace information it collects from executing concurrent programs to improve their performance in subsequent executions.

#### REFERENCES

1. F. A. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing* 5, 617–640 (1988).
2. M. F. Awdeh, OOPS5 – an object oriented production rule system. Technical report, University of Bath, Concurrent Processing Research Group (1989).
3. C. Burdorf and S. Cammarata, PSE: a CLOS-based persistent simulation environment with prefetching capabilities. In *Proceedings of the CLOS Workshop* (1989).
4. C. Burdorf, J. P. Fitch and J. B. Marti, Minimising interprocessor computation overhead. Accepted for publication.
5. C. Burdorf, J. P. Fitch, J. B. Marti and J. A. Padget, A multiprocessor execution profiler. In *Proceedings of 22<sup>nd</sup> Annual Hawaii International Conference on System Sciences*, pp. 524–531. IEE (1989).
6. C. Burdorf and J. B. Marti, Load balancing strategies for time warp on multi-user workstations. In preparation (1989).
7. N. Carriero and D. Gelernter. Linda in context. *Comm. ACM* 32 (4); 444–459 (1989).
8. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck. An efficient method of computing single static assignment form. In *Proceedings of Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 25–35. ACM (1989).
9. J. P. Fitch, How can REDUCE be run in parallel? In *Proceedings of ISSAC'89*, pp. 155–162. ACM (1989).
10. J. P. Fitch and J. B. Marti, The Bath concurrent Lisp machine. In *Proceedings of EUROCAL '83*, vol. 162 of LNCS, pp. 78–90. Springer-Verlag (1984).
11. J. P. Fitch and J. B. Marti, The static estimation of run time. Technical report, University of Bath Computing Group (1987).
12. L. Forgy, OPS5 user's manual. Technical Report CMU-CS-80-13, Department of Computer Science, Carnegie-Mellon University (1981).
13. R. P. Gabriel and J. M. McCarthy, Queue-based multiprocessing lisp. In *Proceedings of 1984 ACM Conference on Lisp and Functional Programming*. ACM (1984).
14. B. L. Gates and J. B. Marti, An empirical study of time-warp systems. In *Proceedings of the Winter Simulation Conference* (1988).
15. N. Graube. *Architectures réflexives et implémentations des langages à taxonomie de classes en Lisp: Applications à ObjVlisp, Common Lisp Object System et TEAOΣ*. PhD thesis, l'Université Paris 6 (1989).
16. M. L. Griss, E. Benson and G. Q. Maguire, PSL: a portable LISP system. In *Proceedings of 1982 ACM Symposium on LISP and Functional Programming*. ACM (1982).
17. A. Gupta, *Parallelism in Production Systems*. PhD thesis, Carnegie-Mellon University (1987).
18. A. Gupta and H. G. Okuno, Parallelising production

#### 5. CONCLUSION

This paper cannot really have a conclusion, since it only describes some steps along the way to a distant goal. What we believe we have established so far is that different concurrent processing primitives can co-exist and can be implemented fairly efficiently even on ordinary workstations connected on a local area network. We are now in the process of transferring this to a loosely-coupled system of tightly-coupled vector processors (that is, two multi-processor Stardent systems connected by ethernet). Our work on analysing Lisp programs has produced meaningful and usable results.

We conjecture that no single concurrent operator is ideal for all granularities of parallelism and that the mixture we have captures a useful selection of granularities. We also conjecture that extracting concurrency from existing and new applications, written without any particular regard for concurrency is tractable and that, in a few years, semi-automatic translators from sequential to concurrent programs are feasible.

#### Acknowledgements

As must be obvious from the breadth of the topics covered in this paper, this is the work of a large number of people. Acknowledgements are due to the other members of the Concurrent Processing Research Group (CPRG) at Bath: Mohammed Awdeh, James Davenport, Dave De'Roure, Nuong Quang Dinh, David Hutchinson, Spiridon Kalogeropoulos, Keith Playford and Icarus Sparry and to the Concurrent Processing for Advanced Simulation group (CPAS) at the RAND Corporation, Santa Monica: Christopher Burdorf, Barbara Gates, Tony Hearn and Jed Marti.



- systems. Technical report, Stanford University Computer Science Department, (1988).
19. R. H. Halstead, Multilisp: a language for concurrent symbolic computation. *ACM TOPLAS* 7, 501-538, (1985).
20. A. C. Hearn. *The Reduce Manual*. The RAND Corporation (1988).
21. D. J. C. Hutchinson, Implementing futures and linda. Technical report, University of Bath, Concurrent Processing Research Group (1989).
22. D. Jefferson. Virtual time. *ACM TOPLAS* 7, 404-425 (1985).
23. S. Kalogeropoulos, Partitioning Lisp programs for parallel execution. Technical Report TR-89-28, University of Bath Computing Group (1989).
24. S. Kalogeropoulos, *The Static Analysis of Lisp Programs for Parallel Execution*. PhD thesis, University of Bath (1990).
25. D. A. Kranz, R. H. Halstead and E. Mohr, Mul-T: A high-performance parallel Lisp. In *Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 81-90. ACM (1989).
26. J.-J. Levy, private communication (1990)
27. J. B. Marti. *The Hasty Evaluator*. PhD thesis, University of Utah (1978).
28. H. Melenk and W. Neun, Parallel polynomial operations in the Burchberger algorithm. In *Computer Algebra and Parallelism*. Academic Press (1989).
29. J. S. Miller, *MultiScheme: A Parallel Processing System*. PhD thesis, Massachusetts Institute of Technology (1987).
30. E. Mohr, D. A. Kranz and R. H. Halstead, Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of 1990 ACM Conference on Lisp and Functional Programming*. ACM (1990).
31. J. A. Padget and A. Barnes, Univariate power series expansions in Reduce. In *Proceedings of ISSAC'90*. Addison-Wesley (1990).
32. J. A. Padget and G. Nuyens (eds), *The EuLISP definition (version 0.69)*. Technical report, University of Bath Computing Group (1990).
33. J. Piquet, Multi-processus en Le-Lisp: Pive II. Rapport de Stage, Université de Paris XI, Orsay (1988).
34. M. Radlhammer, The future of futures, or, how futures can be implemented on stock hardware. In *BCS High Performance and Parallel Lisp Workshop*. EUROPAL (1990).
35. L. A. Rowe, A shared object hierarchy. In *International Workshop on Object-Oriented Database Systems*. IEEE (1986).

## Book Review

RANALD ROBERTSON

*Legal Protection of Computer Software*. London: Longman Law Tax and Finance, 1990. ISBN 085121 6846. Price £35.00.

This is a well-researched and well-presented book covering all aspects of the subject from contractual protection, through copyright, trade secrets and trade marks to patent protection. There is a very useful table of cases, and the book ends with a section on the remedies available should any of the rights being protected be infringed.

Primarily it is a book for lawyers and expert witnesses, but it is readable enough that a computer professional would find little difficulty in following it, and would find much of it instructive. Two important matters are dealt with in appendices, namely a form of contract guidelines for program licensing prepared by the CSA, and a guide to the position relating to protection of software in Europe. Both of these are too short to contain the quality of information provided in the main text. One might hope that later editions will include more detailed coverage of the European situation, since EEC directives are having an increasing influence in UK courts. Contractual matters are, I understand, more fully dealt with in other Longman publications.

I do not recommend all computer professionals to go out and purchase a copy immediately. However, if they become involved in matters of legal protection of their software, and particularly if they are freelancers who wish to retain their rights, it is very valuable reading and explains clearly how the law affects them. For those with a greater interest in the law in this area, whether computer professionals or lawyers, it is undoubtedly strongly recommended reading.

A. S. DOUGLAS  
London

GORDON HUGHES (editor)

*Essays on Computer Law*. London: Longman Group UK Ltd, 1990. ISBN 0 582 93991 7. Price £39.00.

In the Foreword, the Governor-General of Australia, Sir Ninian Stephen, says 'A remarkable feature of this collection of essays is the great diversity of its themes' and goes on to remark 'Through the pages of this volume computers can be seen both as useful tools of legal education and practice of the law and, as themselves, formidably at work in changing the whole legal environment; creating new and important relationships and subject matters, with which the law must come to terms'.

As will be seen from the foregoing, this is an ambitious work, whilst the background of the contributors links it to the Common Law as practised 'down under'. This, of course, means that it is largely relevant to English law, although not directly aimed at it, but only confronts the situation in the EEC occasionally, as in the contribution of Michael Kirby on Trans Border Data Flows, where he has had an important influence on our thinking through his involvement with the OECD.

It is inevitable that on a canvas so large as this, the coverage of detail is often variable. Certainly the treatment will be criticised by readers depending on their background knowledge. If Colin Tapper wrote less interestingly, lawyers could probably skip his introduction. Computer people probably don't need to read Mr Burnside's introduction to computers. UK practitioners, legal or technical, will have little interest in Section F on Taxation, which relates wholly to Australian peculiarities. Similarities with, and differences from, the UK position are well brought out in the sections on Intellectual Property, Data Protection, Crime and the Supply of Computer Products. Some parts of these have general application - I have already mentioned Trans Border Data Flows, which come under Data Protection, and should draw attention also to another essay in that section on Computer Security, which appears better balanced than

many offerings from 'security consultants' intent on selling their latest gimmick for making management feel happy, whilst not significantly decreasing their risk.

Section G on Evidence and Court Proceedings is of general interest. The article on Admissibility of Computer Output reviews the problems without getting entangled in the matters which have incited Colin Tapper elsewhere to remark that the legal position in the UK is of 'Byzantine complexity' - perhaps Australia has escaped this fate! The following article on Alternative Dispute Resolution indicates that matters have advanced further in some Australian states than they have in this country. The lawyers there appear to be more friendly to attempts to simplify the settlement of disputes other than by litigation than is the case in the UK, and to be already following the US lead in this matter, whereas only tentative moves are being made here.

Section H on Practical Uses of Computers has a strong Australian flavour, and is mainly of interest in comparing what is going on there with what is on the agenda here and elsewhere. It is followed by a series of essays on progress (or lack of it) in other countries, including New Zealand, Malaysia, Singapore, the USA, Canada and South Africa, and a summary of how the Europeans and Japanese have approached similar problems. It is interesting to find an article in this section by Philippa Perry reviewing developments in the UK, a good deal of which has been overtaken by the rapidly moving events in the last year or so.

Overall there is much to commend these essays to someone wishing to get an overview of the legal situation in Australia and to compare this with the situation in the UK. As I have indicated, some of the essays are deserving of a wider audience, being of greater generality. I believe this book should be in any library relating to computers and the law, but I could not wholeheartedly recommend it for purchase by students in the UK.

A. S. DOUGLAS  
London