Conclusions and Directions for Future Work

Because of problems like deadlock and livelock, parallel programs are significantly more difficult to design than serial ones. Perhaps for this reason concurrent programming has been slow to take off. Our hunger for computing power is largely being satisfied by the continual development of ever faster serial processors. However there are limits to serial hardware technology that are likely to be approached within the next twenty years. Explicit parallelism will then become the only means of extending the performance of computers and the field of concurrent programming will finally have come of age.

This thesis has described a collection of simple design rules for constructing large scale parallel systems that can never deadlock. We have also detailed efficient techniques for the machine verification of adherence to these rules. More interestingly, a technique for efficiently proving deadlock-freedom was discovered which, despite having no intelligence regarding the design rules, was found to be capable of proving deadlock-freedom for networks constructed according to the majority of them. However it is important to note that this algorithm, which is called CSDD, is far from a complete proof technique for deadlock-freedom. There are many deadlock-free networks which it cannot prove to be so. It works by checking a stronger property than deadlock-freedom – one that can be established in $O(n^2)$ time complexity for networks of finite-state processes, being based purely on local analysis. This compares favourably with the exponential complexity of using FDR for deadlock analysis. On the other hand, the FDR approach is complete for finite-state networks.

The CSDD algorithm is sufficiently simple that it seems suitable for inclusion in compilers for high-level parallel languages such as occam. However to make this feasible, we need to look at methods to restrict the size of the state-spaces to be analysed.

A tool to verify the validity of an *abstraction*, or indeed to perform abstraction automatically would be very useful. Recall that abstraction is the act of replacing detailed communication events by their channel names in CSP networks. In effect, this means 'throwing away the data'. This technique has been used throughout the thesis to simplify CSP expressions to be analysed for deadlock-freedom. In this way, infinite-state networks may be proven deadlock-free by the analysis of finite state abstractions. A formal statement of the property is given in [Roscoe 1995].

Without the use of abstraction, the operational representation of even a very simple occam process might be vast. A technique for conversion from occam to CSP is described in [Scattergood and Seidel 1994] which addresses this problem.

The deadlock analysis techniques that we have described are based on a static network of non-terminating processes grouped together by a single level of parallelism. We rely on each process having a relatively small number of states. Efficient deadlock analysis is then possible because we avoid constructing the state transition system for the network as a whole. However if there were a large degree of embedded parallelism in any component process of the network, we would still need to analyse some unwieldy operational representations.

A good illustration of this problem is a program from [Jones and Goldsmith 1988] which implements Conway's Game of Life using an array of *I/O-PAR* processes, each with 16 channels. The fact is that the abstract operational form of each of these harmless looking processes has 65536 states. This would certainly make the pairwise process checking performed by the CSDD algorithm impractical.

It should be possible to develop transformational techniques to cope with networks like this which remove all the embedded parallelism to the outer layer. For instance, consider the *I/O-SEQ* process

$$P = (a \rightarrow \mathit{SKIP} \mid\mid\mid b \rightarrow \mathit{SKIP}) \; ; (c \rightarrow \mathit{SKIP} \mid\mid\mid d \rightarrow \mathit{SKIP}) \; ; P$$

We transform this process into a subnetwork of five purely sequential processes as follows.

$$\begin{array}{lcl}
P' & = & s_1 \rightarrow s_2 \rightarrow f_1 \rightarrow f_2 \rightarrow s_3 \rightarrow s_4 \rightarrow f_3 \rightarrow f_4 \rightarrow P' \\
Q_1 & = & s_1 \rightarrow a \rightarrow \mathit{SKIP} \; ; f_1 \rightarrow Q_1 \\
Q_2 & = & s_2 \rightarrow b \rightarrow \mathit{SKIP} \; ; f_2 \rightarrow Q_2 \\
Q_3 & = & s_3 \rightarrow c \rightarrow \mathit{SKIP} \; ; f_3 \rightarrow Q_3 \\
Q_4 & = & s_4 \rightarrow d \rightarrow \mathit{SKIP} \; ; f_4 \rightarrow Q_4
\end{array}$$

Events s_i and f_i are 'start' and 'finish' commands for each subprocess Q_i, sent out by the master process P'. The following equivalence may be shown (using FDR).

$$P = \mathit{PAR}(\langle P', Q_1, Q_2, Q_3, Q_4 \rangle) \setminus \{s_i, f_i | i = 1, 2, 3, 4\}$$

We could use this as follows. First we could prove divergence-freedom for P by using Deadlock Checker to prove livelock-freedom for

$$\mathit{PAR}(\langle P', Q_1, Q_2, Q_3, Q_4 \rangle)$$

That would allow us to substitute $\{P', Q_1, Q_2, Q_3, Q_4\}$ for P into any network to be tested for deadlock-freedom.

Similarly transforming an *I/O-PAR* process with 16 channels, such as used in the Jones and Goldsmith program, would result in a subnetwork with seventeen processes, each one being a purely sequential cyclic process with just a handful of states. This would provide a representation suitable for analysis by Deadlock-Checker.

It would be very useful to explore general situations where such transformations might be applied. This approach ought to be particularly applicable to occam programs, where it is common to have several layers of embedded parallelism.

Deadlock-freedom is only the tip of the iceberg when it comes to proving desirable properties of concurrent systems. If we were to design a signalling system for trains based on these methods it would certainly be a good idea to prove the system deadlock-free, but it would be somewhat more important to ensure that no two trains could ever collide. The FDR tool can be used to do proofs like this by exhaustive state analysis. But due to the exponential state explosion as a network grows in size this method cannot be used for very large networks. Certainly not the Great Western Railway network.

Specifications which prohibit undesirable actions are known as *safety conditions*, and are generally expressed purely in terms of traces, refusal sets being irrelevant. One approach to proving safety properties of large systems is to factorise the proofs into smaller manageable parts. In order to prove that no two trains can ever collide we might attempt to prove separately a large number of statements of the form: "*TRAINA* and *TRAINB* will never collide on the track section governed by *SIGNAL1*". If we could show that this statement held true for the subnetwork

$$\langle \mathit{TRAINA}, \mathit{TRAINB}, \mathit{SIGNAL1} \rangle$$

then clearly it would hold for the the network as a whole. This could be done using the refinement checker FDR as the number of states of the subnetwork ought to be manageable. There is scope for developing a logical inference tool to assist with proofs of this kind.

It is also common to write specifications which insist that some desirable form of behaviour should occur. For instance, we might specify that the electric doors of a train's carriages should never refuse to be opened when the train is standing at a platform. Specifications such as this are called *liveness conditions* and they require the full expressive power of the failures model.

Dathi's thesis [Dathi 1990] contains the attractive idea of transforming a general failures specification problem into a proof of deadlock-freedom. Given a concurrent system $V = \langle P_1, .., P_n \rangle$ and a specification S we want to show the refinement relation

$$\mathit{failures}(S) \supseteq \mathit{failures}(\mathit{PAR}(V) \setminus (\alpha V - \alpha S))$$

Dathi defines a process transformation function δ so that proving the refinement reduces to showing that the network

$$\langle \delta(S), P_1, .., P_n \rangle$$

is deadlock-free. Basically $\delta(S)$ is a 'testing' process which guarantees to deadlock the network whenever $\mathit{PAR}(V) \setminus (\alpha V - \alpha S)$ exhibits any behaviour which is illegal for S. Unfortunately the process $\delta(S)$ is not itself deadlock-free so we cannot use any of the local analysis techniques described in this thesis to prove the refinement. However Dathi defines a similar transformation function δ^* which produces better behaved processes $\delta^*(S)$. It this case it is first necessary to prove, by other means, that

$$\textit{traces}(S) \supseteq \textit{traces}(\textit{PAR}(V) \setminus (\alpha V - \alpha S))$$

We may then show that the failures specification is satisfied by proving the network

$$\langle \delta^*(S), P_1, .., P_n \rangle$$

to be deadlock-free. It should be straightforward to automate this technique for inclusion in a tool like Deadlock Checker. Design rules might then be formulated for the type of specifications that could be checked.

A different approach is likely to be required when dealing with issues relating to the correctness of computation rather than communication. Recalling the program to solve Laplace's equation in chapter 4, we have proven that this program cannot deadlock, but we are yet to show that it accurately calculates a solution to the problem. The prototype CSP code does not contain enough information to do this. We need to consider the refinement into the final occam version. For this program, any conventional operational representation would be vast. In order to construct it we would effectively have to perform the entire computation for every single possible variety of initial conditions. Ideally what is required is a two-tiered form of operational semantics so that information regarding computation is represented on a separate level from information regarding communication.

Another important issue that has not been considered in this thesis is time. A major motivation for parallel computation is speed of results. Therefore we are likely to have hard real-time requirements for the systems that we design. If an airman presses the button to switch off the autopilot he should not have to wait for half an hour for anything to happen. The state of the art regarding the use of timed CSP is described in [Davies 1993]. A complete method for proving adherence to timed specifications is presented, but the author recognises that the large number of proofs required for applications of a significant size is likely to be infeasible. Therefore it would seem that there is a need for design rules to be discovered which would facilitate the development of real-time systems. Perhaps the most promising of the design rules considered here, from this point of view, is the cyclic paradigm. The processes could be synchronised to operate with a computation phase and a communication phase of fixed time, say δt. The time for various external operations to be effective should then be easy to predict as a multiple of δt. This idea is similar to the BSP paradigm of L. Valiant (documented in [Oxford Parallel 1995]).

A Vision for the Future

It would seem that there is still some work to be done before the construction of large-scale parallel programs can be regarded as a thoroughly safe engineering discipline.

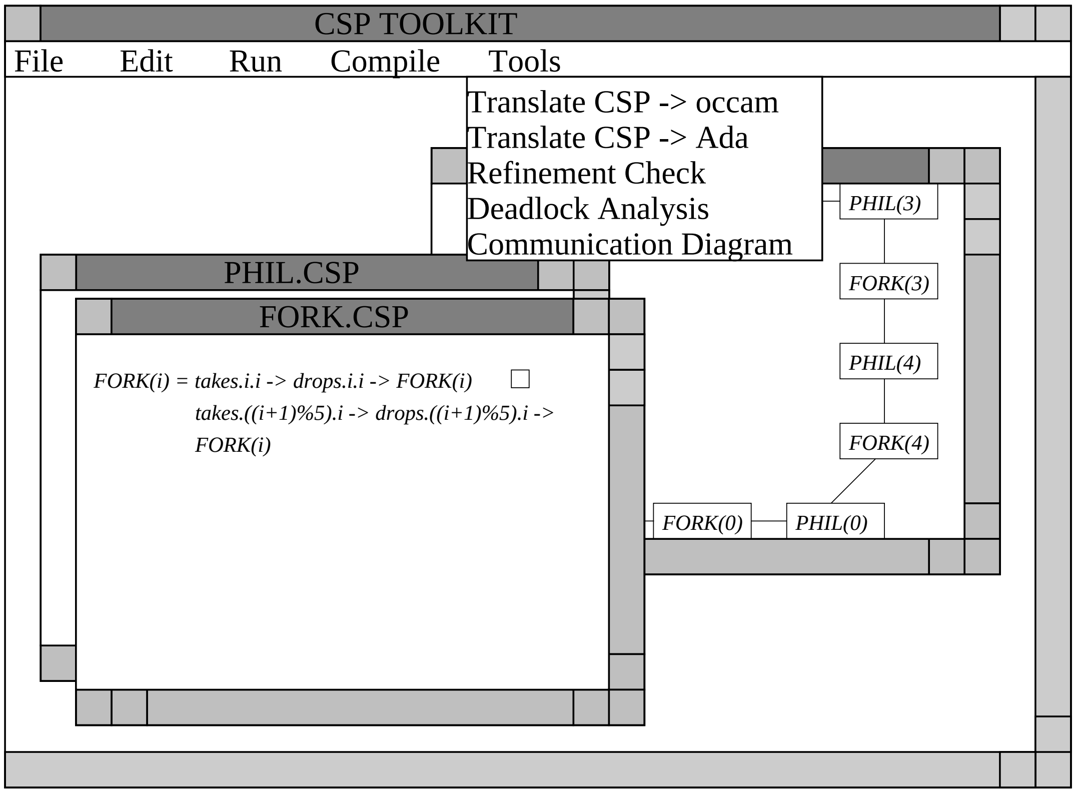
Hopefully this thesis has outlined an approach to one of the major problems which is of clear practical use. Ease of use and simplicity of presentation have to be major goals in developing tools for engineers.

In the near future, there is the exciting prospect of an integrated CSP development environment, such as illustrated in figure 4.6. Programs could be systematically refined from their abstract specifications making use of a number of tools, to perform such functions as refinement checking, abstraction checking, real-time specification checking, conversion to and from high-level programming languages, and, of course, deadlock and livelock analysis. In the event of a potential deadlock being detected the tool would be able to point back to the exact position in the source of each process involved in it.

CSP is a most elegant language and it would be nice if it could be used directly for actual programming, rather than having to convert to another language such as occam. Then we could use the same notation all the way through from specification to implementation. In order to make an efficient CSP compiler, one would need to enforce certain restrictions, such as the restriction in occam that external choice can only be applied to input channels. There would also need to be some thought applied over the treatment of the state of variables and their scope. Probably the language that we would finally arrive at would be functionally very similar to occam, but it would look like CSP. Of course some people might not consider the CSP notation to provide the most *readable* presentation style for concurrent software, preferring the use of words to symbols. There is no reason why a verbose isomorphism of CSP should not be provided for such people, rather than a different language.

By making life easier for engineers we might reduce the potential for software-precipitated catastrophes. But it is very important that we always maintain a clear view of the limitations of formal methods. For instance, they cannot guard against a leaky specification which fails to incorporate vital safety information. There will always be a human decision-making aspect to software construction. By making the programming environment helpful, intuitive and secure we can help to ensure that the right decisions are made.

Figure 4.6: CSP Toolkit – A Vision for the Future



```
FORK(i) = takes.i.i -> drops.i.i -> FORK(i)      □
          takes.((i+1)%5).i -> drops.((i+1)%5).i ->
          FORK(i)
```