Deadlock Free Specification Based on Local Process Properties

D.P.Simpson and J.S.Reeve

The Department of Electronics and Computer Science,
University of Southampton, Southampton SO17 1BJ, UK
email jsr@ecs.soton.ac.uk

Abstract. We present a design methodology for the construction of parallel programs that is deadlock free, Provided that the "components" of the program are constructed according to a set of locally applied rules. In our model, a parallel program is a set of processes and a set of events. Each event is shared by two processes only and each process progresses cyclically. Events are distinguished as input and output events with respect to their two participating processes. On each cycle a process must complete all output events that it offers to the environment, be prepared to accept any, and accept at least one, of its input events before completing any computations and starting a new cycle. We show that however the events are distributed among the processes, the program is deadlock free. Using this model we can construct libraries of constituent processes that do not require any global analysis to establish freedom from deadlock when they are used to construct complete parallel programs.

1 Introduction

In this paper we develop a programming design methodology for parallel programs that manifestly avoids deadlock. Deadlock is an issue in many different types of program and examples of explicit attempts at producing or establishing deadlock free formulations can be found in applications ranging from robotics[12] and control[6], to scheduling[1, 7] and graph reduction[19]. More general methodologies[2, 11] have also been published. The programming strategy that we establish in this paper is a general method of constructing algorithmically parallel[16] programs so that they cannot deadlock. Our methodology comes in two flavours, one is a synchronised model which is guaranteed to produce "live" programs but at the cost of some redundant communications. The other, a non-synchronised model, removes the event redundancy at the cost of "liveness". In practice however, as we show in §5 liveness is restored by a fair queueing scheme.

This paper shows how to structure and combine components, using only 'local' or component centred rules to ensure that the resulting program is free from deadlock, even with arbitrary cycles in the communication graph. No global requirements, other than each output must have a corresponding input, are necessary, and no overall synchronization is needed. The programming model is readily implementable with existing software, for example MPI [20].

A program is composed of components which cycle continuously. Each component is structured in pseudo code as

```
y = y_0 INITIAL OUTPUT VARIABLES
x = x_0 INITIAL INPUT VARIABLES
FOREVER
    DO IN PARALLEL
        OUTPUT y
        INPUT AT LEAST ONE x
    END DO
    COMPUTE y = f(x)
```

The details of the computation are application dependent, but we assume that it comprises actions internal to the local process only and has no bearing on the event sequence of the overall program.

Deadlock due to components waiting to send a message before they are willing to receive one is impossible because wanting to send a message implies being willing to read one, unless a message has been read from that source in this cycle already. Consequently closed cycles don't appear in the communication graph[13]. Furthermore, no fairness conditions, which specify "equal treatment" conditions, of any kind are required. Components can compute an appropriate subset of their output to exercise, which can be different on each cycle, without invalidating the result. We call this the asynchronous model. A related model ,the synchronous model, which also has the same freedom from deadlock, exercises all its output on every cycle.

In practice the asynchronous model is more efficient as fewer messages are being passed around the system but the synchronous model has long been used to ensure deadlock freedom in Petri-Net designs [5, 9] or other state transition based methods[3, 14, 15].

Figure 1 is a pictorial representation of a particular parallel program, the alphabet of which is $\{a, b, c, d, e, f\}$ and the constituent processes are $\{P_1, P_2, P_3, P_4, P_5\}$. This program can operate according to our asynchronous or synchronous rules, but not a mixture of both. The alphabet is partitioned into input events and output events with respect to each constituent process. For example the input event set (depicted by the inward arrows) for process P_1 is $I_1 = \{d, f\}$ and the output event set (depicted by the outward arrows) is $O_1 = \{a\}$. Similarly, $I_2 = \{g\}$ and $O_2 = \{e, f\}$, and so on. The precise definition of components is written in CSP. The proof establishes that it is impossible for any output to be permanently blocked. In §2 we give some background to Communicating Sequential Processes, CSP, the underlying methodology that we use, and in §3 and §4 we formally define the synchronous and asynchronous models respectively and establish the terms and conditions of their freedom from deadlock. Finally in §5 we show how these models are effectively used in practice.

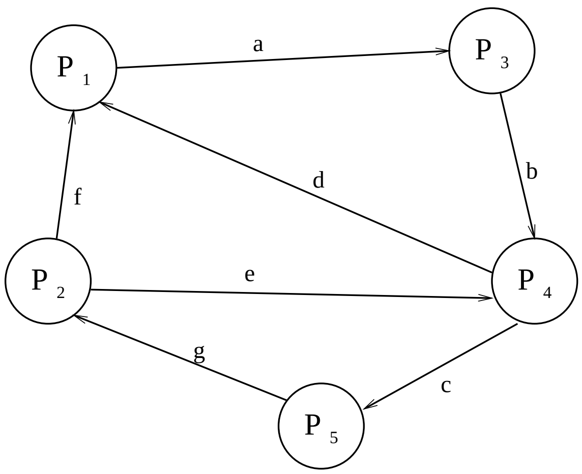

Fig. 1. A Diagrammatic Representation of a Program

2 Theoretical Background

In this paper we use the failures model of Communicating Sequential Processes (CSP) to formally define and verify our programming model . This is a process based model that focuses on the communications events that occur between processes. These events are assumed to be atomic and instantaneous and for an event a to happen, it must be engaged in simultaneously by all processes that contain a in their alphabet. The principle introductory references to CSP are the books by Hoare [8] and Roscoe [17].

A process P is a 3-tuple $(\alpha P, \mathit{failures}(P), \mathit{divergences}(P))$. The alphabet of P, αP, is the set of events that P can engage in . The list of events that a process has engaged in, is its *trace*, and the set of all possible traces of P is $\mathit{traces}(P)$. $\mathit{failures}(P)$ is a set of pairs, (s, R), such that P can engage in the action of $s \in \mathit{traces}(P)$ in order and then refuse to engage in all the actions in the *refusal* set R. After engaging in a trace $t \in \mathit{divergences}(P)$, P can engage in an unbounded sequence of events from its alphabet. This is *livelock*.

P can only affect events in its alphabet and in particular $P \parallel Q$ must synchronize on events in their common alphabet. After engaging in the trace t, P becomes P/t.

In the definitions in the following sections we make use of the special process, RUN, which is a process that never diverges nor refuses any event in its alphabet. This paper uses RUN to eliminate the effect of terminating processes which have done their job via the identity $RUN \parallel P = P$ for any process P provided that $\alpha RUN \subseteq \alpha P$. We also use the result $RUN_A \parallel RUN_B = RUN_{A \cup B}$, which is

readily proven by establishing the equivalence of the *alphabets*, *divergences* and *failures* of the processes ($RUN_A \parallel RUN_B$) and $RUN_{A \cup B}$.
Practical parallel programming processes communicate by passing messages or sharing data at some common location. CSP is not concerned with the 'direction' in which the data is flowing, only in the fact that all the processes with a given event in their alphabet are synchronised when that event happens. Later in this paper we will have to distinguish between *input* and *output* processes because events common to two processes are handled differently by those sharing processes.
The following standard[18] definition means that for our programming model we must establish that the *refusals* set is never the entire alphabet in order to prove our assertion of deadlock freedom.

Definition 1 (Deadlock Freedom) *A process P is deadlock free, if and only if $\nexists\, t \in traces(P)$ such that $(t, \alpha P) \in \mathrm{failures}(P)$.*

The definitions and proofs that establish deadlock freedom for our programming model in this paper are easily established, since the components of the model are constructed to do so. This means that programs constructed according to our prescription do not require a global analysis to establish deadlock freedom as do more general programming models[17].

3 The Synchronous Programming Model

We begin with a simplified model which we call the synchronous programming model. The characteristics of this model are that each component process (node in the graphical representation) executes cyclically and on each cycle engages in all its *input* and *output* events before computing new output variable values. Since every *output* and *input* event is allowed to happen in parallel there are no cycles of event dependency and so deadlock is manifestly not possible. Moreover, every event happens on every cycle. This is the standard way in which deadlock is avoided in state machine driven embedded systems [3].
In the following section we will relax the condition that every input event be satisfied on every cycle. This, while retaining the freedom from deadlock property, eliminates messages(in real systems) that are only there to establish that freedom from deadlock. This relaxation does however, introduce the possibility of some events being refused for an indeterminate number of cycles. In practice though, simple fairness implementations, like FIFO queueing [4] of messages at the constituent processes readily rectifys this.
A synchronous program SP is a finite collection of constituent processes $\{S_1, S_2, \ldots S_n\}$ that interact through a set of events IO. The event set IO has two partitions $\pi_I(IO) = \cup_{j=1,n} I_j$ and $\pi_O(IO) = \cup_{j=1,n} O_j$. I_j and O_j are termed the *input* and *output* events of S_j respectively. In addition each element $io \in IO$ is contained in the alphabet of two, and only two, processes. Hence $\forall\, io \in IO\ \exists$ unique I_i and O_j such that $O_j \cap I_i = \{io\}$

With these definitions in mind we define a synchronous program SP as

$$
\begin{aligned}
SP &\equiv \|_j\ S_j \\
S_j &\equiv OUT_j \parallel IN_j \parallel CYCLE_j
\end{aligned} \tag{1}
$$

Each of the constituent processes S_j has a synchronising event s_j in its alphabet, which facilitates the cycling of the process once all *input* and *output* events have happened. Each S_j then has the structure:-

$$
\begin{aligned}
OUT &\equiv \|_{o\in O}\ o \rightarrow RUN_s \\
IN &\equiv \|_{i\in I}\ i \rightarrow RUN_s \\
CYCLE &\equiv \Box_{io\in O\cup I}\ io \rightarrow CYCLE \ \Box\ s \rightarrow S
\end{aligned} \tag{2}
$$

To accommodate sources and sinks we use the following special definitions when some or all of the constituent event sets are empty.

$$
\begin{aligned}
OUT &\equiv RUN_s \text{ when } O = \{\} \\
IN &\equiv RUN_s \text{ when } I = \{\} \\
CYCLE &\equiv s \rightarrow S \text{ when } O \cup I = \{\}
\end{aligned} \tag{3}
$$

The complete alphabet of S is $O \cup I \cup \{s\}$. The internal computation phase is assumed to happen "between" the s and S in the term $s \rightarrow S$ in the $CYCLE$ process. The *output* process OUT cannot engage in s until all *output* events have happened. Likewise, the *input* process IN cannot engage in s until all *input* events have happened.
We now establish that the OUT and IN processes must engage in all their events before engaging in the synchronising event s. In the following we use the notation $<[O]>$ to stand for a *trace* comprised of all the elements of O in some (unspecified) order.

Lemma 1 *All output events happen once before synchronisation. Provided that the environment is prepared to engage in the events of O, an output process, OUT as defined in equation 2 has a trace, $t = <[O], s>$.*

Proof *If $O = \{\}$ then $OUT = RUN_s$ and the lemma holds since $t = <s>$. When O is not empty,*

$$OUT = \|_{o\in O}\ o \rightarrow RUN_s$$

Because the constituent parallel processes have disjoint alphabets each can engage in its initial event before being ready to engage in s. OUT can engage in the initial events c in any order so any trace of OUT is of the form $t = <[O], s>$.

We note that the above result is a statement about the *trace* of OUT under the assumption that none of the events of O are in the refusal set of any other process. Also because the form of IN is the same as that of OUT, an equivalent result holds, namely:-

Lemma 2 *All input events happen once before synchronisation Provided that the environment is prepared to engage in the events of* I*, an input process,* IN *as defined in equation 2 has a trace,* $t = <[I], s>$.

Now we must show that a synchronous program SP provides the environment for lemma 1 and lemma 2 to hold, which will allow us to establish the cyclical nature of synchronous programs, prove that they are free from deadlock and moreover, show that they are "live", in the sense that every event happens on every cycle.

Theorem 1 *A synchronous program is deadlock free*
If SP *is an asynchronous program as defined in equation 1, then*
$\nexists\, t \in traces(SP)$ *s.t.* $(t, \alpha SP) \in failures(SP)$

Proof *The result required is that the* $refusals(SP) \subset \alpha SP$ *and we note that* $refusals(P_1 \parallel P_2) = refusals(P_1) \cup refusals(P_2)$*[8].*

$$\begin{aligned} refusals(SP) &= \cup_j refusals(S_j) \\ &= \cup_j (refusals(OUT_j) \cup refusals(IN_j) \cup refusals(CYCLE_j)) \end{aligned}$$

The refusals set $refusals(SP)$ *is maximal when* $refusals(S_j)$ *are maximal* $\forall\, j = 1, \ldots, n$. $refusals(OUT_j)$ *and* $refusals(IN_j)$ *are maximal after all the* O_j *and* I_j *have happened, in which case:-*

$$\begin{aligned} &refusals(S_j / <[I_j \cup O_j]>) = O_j \cup I_j \qquad and \\ &refusals(CYCLE_j / <[I_j \cup O_j]>) = \{\} \end{aligned}$$

At this point only events in $SY = \{s_j \mid j = 1, \ldots n\}$ *can happen. However*

$$\begin{aligned} &OUT_j / <[O_j]> = RUN_{s_j} \\ &IN_j / <[I_j]> = RUN_{s_j} \qquad and \\ &CYCLE_j / <[I_j \cup O_j]> = CYCLE_j \end{aligned}$$

Consequently $S_j / <[I_j \cup O_j], s_j> = S_j$ *using lemma 1 and lemma 2 and so* $SP / <[IO], [SY]> = SP$*, and the maximal refusals set of* $SP \subset \alpha SP$.

4 The Asynchronous Programming Model

We now seek to remove some of the restrictions of the synchronous model and in particular consider systems the component processes of which need not exercise all possible output events on every cycle. Our overall definition of an asynchronous program is the same as that of a synchronous program but the component processes behave differently.
Similar to a synchronous program, an asynchronous program AP is a finite collection of constituent processes $\{A_1, A_2, \ldots A_n\}$ that interact through a set of events IO, partitioned as in §3.

We define an asynchronous program AP as

$$\begin{aligned} AP &= \|_j\ A_j \\ A_j &= OUT_j \parallel IN_j \parallel CYCLE_j \end{aligned} \tag{4}$$

Each component component process has the form:

$$\begin{aligned} OUT &= KEY \parallel_{o' \in O'} o' \rightarrow RUN_s \\ KEY &= \parallel_{o^m \in O \setminus O'} (o^m \rightarrow RUN_s \,\Box\, RUN_s) \\ IN &= \parallel_{i \in I} (i \rightarrow RUN_s \,\Box\, RUN_s) \parallel LOCK \\ LOCK &= \Box_{i \in I} \left(i \rightarrow RUN_{I \cup \{s\}}\right) \\ CYCLE &= \Box_{io \in O \cup I} (io \rightarrow CYCLE) \,\Box\, s \rightarrow A' \end{aligned} \tag{5}$$

To accommodate sources and sinks we use the following special definitions when some or all of the constituent event sets are empty.

$$\begin{aligned} OUT &\equiv RUN_s \text{ when } O = \{\} \\ IN &\equiv RUN_s \text{ when } I = \{\} \\ CYCLE &\equiv s \rightarrow A' \text{ when } O \cup I = \{\} \end{aligned} \tag{6}$$

A is the overall component and s synchronises the various component processes. The complete alphabet of A is $O \cup I \cup \{s\}$. A computation phase is assumed to occur "between" the s_j and A'_j in the term $s_j \rightarrow A'_j$ but as the details of the computation phase of the cycle are irrelevant they are not modeled here. A process A_j, in general is allowed to select different active set of outputs $O'_j \subseteq O_j$ for each cycle on any basis. The term $s \rightarrow A'$ indicates that the "active" output set O' may be different on the next cycle. The *output* process OUT cannot terminate until all *active output* events have happened.
We now show that this formal definition has the attributes described in the introduction. The IN process ensures that at least one input events completes before IN is ready to engage in the synchronising event s. Now because each process can select the output events that must complete on every cycle, there remains the possibility that a process A_j could become isolated by virtue of not being offered any of its input events by any of the other processes. This scenario is circumvented by the presence of the KEY process in the OUT process.
The following lemma shows that for an output process OUT, all output events happen once and only once before the synchronising event s happens.

Lemma 3 *All output events happen once before synchronisation Provided that the environment is initially prepared to engage in O, an output process*

$$OUT = KEY \parallel_{o' \in O'} o' \rightarrow RUN_s$$

has a trace, $t = <[Q \cup O'], s>$, where $Q \subseteq O \setminus O'$.

Proof *If* $O' = \{\}$ *then* $OUT = RUN_s$ *and the lemma holds since* $t = <s>$. *When* O' *is not empty,*

$$OUT = KEY \parallel_{o' \in O'} o' \rightarrow RUN_s$$

Because the constituent parallel processes have disjoint alphabets each can engage in its initial event before being ready to engage in s. OUT *can engage in the initial events* O' *in any order and* KEY *can engage in any subset* $Q \subseteq O \backslash O'$ *so any trace of* OUT *is of the form* $t = <[Q \cup O'], s>$.

The following lemma shows that for an input process IN, at least one input events happens before the synchronising event s happens. A particular input event can only happen once before s happens

Lemma 4 *At least one input event happens before synchronisation Provided that the environment is initially able to engage in* I, *an input process*

$$IN = \parallel_{i \in I} (i \rightarrow RUN_s \,\Box\, RUN_s) \parallel LOCK$$
$$LOCK = \Box_{i \in I} \left(i \rightarrow RUN_{I \cup \{s\}}\right)$$

has a trace, $t = <[Q], s>$, *where* $Q \subseteq I$.

Proof *If* $I = \{\}$ *then* $IN = RUN_s$ *and the lemma holds since* $t = <s>$. *When* I *is not empty,*

$$IN = \parallel_{i \in I} (i \rightarrow RUN_s \,\Box\, RUN_s) \parallel \Box_{i \in I} \left(i \rightarrow RUN_{I \cup \{s\}}\right)$$

Both the process

$\Box_{i \in I} \left(i \rightarrow RUN_{I \cup \{s\}}\right)$ *and the process* $\parallel_{i \in I} (i \rightarrow RUN_s \,\Box\, RUN_s)$

can engage $Q \neq \{\} \subseteq I$, *then*

$$\Box_{i \in I} \left(i \rightarrow RUN_{I \cup \{s\}}\right) / <[Q]> = RUN_{I \cup \{s\}}$$

and

$$\parallel_{i \in I} (i \rightarrow RUN_s \,\Box\, RUN_s) / <[Q]> = RUN_s \parallel_{i = \in I \backslash Q} (i \rightarrow RUN_s \,\Box\, RUN_s)$$

If s *now occurs we have*

$$RUN_{I \cup \{s\}} \parallel_{i \in I \backslash Q} (i \rightarrow RUN_s \,\Box\, RUN_s) / <[Q], s> = RUN_{I \cup \{s\}}$$

then $t = <[Q], s>$ *and the lemma is established.*

Now we must show that a synchronous program AP provides the environment for lemma 3 and lemma 4 to hold, which will allow us to establish the cyclical nature of asynchronous programs, prove that they are free from deadlock. However asynchronous programs are not “live” in the sense that every event need not happen on every cycle.

Theorem 2 *An asynchronous program is deadlock free*
If AP is an asynchronous program as defined in equation 4, then
$\nexists\, t \in traces(AP)$ *s.t.* $(t, \alpha AP) \in failures(AP)$

Proof *The refusals set $refusals(AP)$ is maximal when $refusals(S_j)$ are maximal $\forall\, j = 1, \ldots, n$. $refusals(OUT_j)$ and $refusals(IN_j)$ are maximal after all the O_j and I_j have happened, in which case:-*

$$refusals(S_j / <[I_j \cup O_j]>) = O_j \cup I_j \qquad and$$
$$refusals(CYCLE_j / <[I_j \cup O_j]>) = \{\}$$

At this point only events in $SY = \{s_j \mid j = 1, \ldots n\}$ can happen. However

$$OUT_j / <[O_j]> = RUN_{s_j}$$
$$IN_j / <[I_j]> = RUN_{s_j} \qquad and$$
$$CYCLE_j / <[I_j \cup O_j]> = CYCLE_j$$

Consequently $A_j / <[I_j \cup O_j], s_j> = A_j$, by lemma 3 and lemma 4 and so $AP / <[IO], [SY]> = AP$, and the maximal refusals set of $AP \subset \alpha AP$.

5 Conclusion

We present a design methodology for the construction of parallel programs that is deadlock free, Provided that the "components" of the program are constructed according to a set of locally applied rules. In our model, a parallel program is a set of processes and a set of events. Each event is shared by two processes only and each process progresses cyclically. Events are distinguished as input and output events with respect to their two participating processes. On each cycle a process must complete all output events that it offers to the environment, be prepared to accept any, and accept at least one, of its input events before completing any computations and starting a new cycle. We show that however the events are distributed among the processes, the program is deadlock free. Using this model then, we can construct libraries of constituent processes that do not require any global analysis to establish freedom from deadlock when they are used to construct complete parallel programs[10].
In practice, parallel program construction according to our asynchronous model, is readily implemented and made more effective by incorporating local fairness. For example the MPI libraries provide means to input all waiting messages from unique sources, thereby assuring fair servicing of inputs while at the same time not allowing two inputs from the same source on the same program cycle.
We conclude that our programming model, although not suited to all possible parallel program communication patterns, particularly where the task graph is dynamic, does ensure deadlock freedom in algorithmic parallelism which is the principle paradigm used in the programming of embedded systems.

References

1. TF Abdelzaher and KG Shin. Combined task and message scheduling in distributed real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 10:1179–1191, 1999.
2. S Abramsky, SJ Gay, and R Nagarajan. A specification structure for deadlock-freedom of synchronous processes. *Theor. Comput. Sci.*, 222:1–53, 1999.
3. A. Arnold. *Finite Transition Systems.* Prentice Hall, Hemel Hempstead, 1992.
4. M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall, Hemel Hempstead, 1990.
5. DY Chao. Petri net synthesis and synchronization using knitting technique. *J. Inf. Sci. Eng.*, 15:543–568, 1999.
6. L Ferrarini, L Piroddi, and S Allegri. A comparative performance analysis of deadlock avoidance control algorithms for FMS. *J. Intell. Manuf.*, 10:569–585, 1999.
7. BP Gan and SJ Turner. An asynchronous protocol for virtual factory simulation on shared memory multiprocessor systems. *J. Oper. Res. Soc.*, 51:413–422, 2000.
8. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Hemel Hempstead, 1985.
9. K. Jensen and G. Rozenberg. *High-level Petri nets: Theory and Applications.* Springer-Verlag, Berlin, 1991.
10. J.S.Reeve and M.C.Rogers. MP: an application specific concurrent language. *Concurrency: Practice anmd Experience*, 8(4):313–333, May 1996.
11. D Kadamuddi and JJP Tsai. Clustering algorithm for parallelizing software systems in multiprocessors environment. *IEEE Trans. Softw. Eng.*, 26:340–361, 2000.
12. T Matsuura and Y Maeda. Deadlock avoidance of a multi-agent robot based on a network of chaotic elements. *Adv. Robot.*, 13:249–251, 1999.
13. T.G. Lewis nad H. EL-Rewini. *Introduction to Parallel Computing.* Prentice Hall, Hemel Hempstead, 1992.
14. TS Perraju and BE Prasad. An algorithm for maintaining working memory consistency in multiple rule firing systems. *Data Knowl. Eng.*, 32:181–198, 2000.
15. A Pnueli, N Shankar, and E Singerman. Fair synchronous transition systems and their liveness proofs. *IEEE Trans. Parallel Distrib. Syst.*, 1486:198–209, 1998.
16. D.J. Pritchard. Practical parallelism using transputer arrays. *Lecture Notes in Computer Science*, 258:278, 1987.
17. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, Hemel Hempstead, 1998.
18. A. W. Roscoe and N. Daith. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, 1987.
19. W Sadiq and ME Orlowska. Analyzing process models using graph reduction techniques. *Inf. Syst.*, 25:117–134, 2000.
20. M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra. *MPI: The Complete Reference.* The MIT Press, Cambridge, Massachusetts, 1996.