Static Analysis of Interprocess Communication[1]

Peter B. Ladkin
ladkin@techfak.uni-bielefeld.de
Universität Bielefeld, Technische Fakultät, D-33501 Bielefeld

Barbara B. Simons
simons@vnet.ibm.com
Applications Development Technology Institute (ADTI)
IBM Software Solutions Division M24/E254
555 Bailey Rd., San Jose, CA 95141

Version of April 22, 1995

[1]This is a version of a book recommended by Fred Schneider and accepted by David Gries to appear in the series **Lecture Notes in Computer Science** published by **Springer-Verlag**. We are revising the manuscript for publication and would be very grateful for any comments.

Contents

Preface

We have been working jointly on the static analysis problems whose solutions appear in this book for a few years. We couldn't seem to encapsulate the basic work in a normal-length journal article, and so decided to collect the results in a monograph.

The work was undertaken while the first author was travelling. Consequently, there are many people and institutions who have in one way or another facilitated the production of this work. He gratefully acknowledges the hospitality of Prof. Dr. Friedrich Vogt and the Deutsche Forschungsgesellschaft for the opportunity to pursue this work while Zuse Gastprofessor at the Universität Hamburg, Prof. Dr. Dieter Hogrefe and the Universität Bern, Dr. Ron Fagin and the IBM Almaden Research Center, Prof. Ken Turner and the University of Stirling, and Prof. Dominique Méry, CRIN-CNRS at the Université Henri Poincaré, Nancy I, and INRIA Lorraine.

We especially thank Dr. Ana M. D. Moreira of the Universidade Nova de Lisboa, and Dr. Ashley McClenaghan, both formerly of the University of Stirling, for help with the graphics. We gratefully acknowledge Prof. Charles Martel and Dr. Ron Fagin for constructive comments, and Russell Brand for some effective programming which originally inspired us to look closer at the problem of static analysis. We thank Prof. Fred Schneider for recommending our work, and the Springer editor, Prof. David Gries. Finally, we avoid a circular reference to Ellen Gilkerson by mentioning her here.

Chapter 1

Introduction

We present algorithms for statically analyzing communications among concurrently executing processes. Our goal is to devise practical methods which can be used in systems that employ message passing. Examples of such systems are distributed memory machines, telecommunications protocols and services, data flow diagrams and process specifications in software development. Because we consider only the class of problems for which deterministic algorithms exist, and because analysis for the general case is intractable, the process definitions must be restricted. In the final chapter we analyze the complexity of the deadlock detection problem and show that relaxing the restrictions even slightly causes the problem to become NP-hard.

Static analysis is quite sensitive to the precise form of the communication primitives used by the processes. We present a detailed description and analysis of a deterministic algorithm for a specific primitive, and then illustrate how to modify the algorithm to accommodate common primitives with different properties. While working on the termination proof for the algorithm, we encountered an interesting combinatorial problem, the analysis of which led to the discovery of the Numerical Matching Condition (NMC). The NMC, together with what we call the Sequence Condition, can be used as stand-alone tests. If either fails, then deadlock is a certainty; if both hold, we cannot guarantee the absence of deadlock without executing one of the algorithms.

1.1 The Problems

Distributed systems, high-level software designs, software for parallel processors, and communications protocols can be specified as collections of communicating processes. It is very hard to analyse software for such systems, whether it is expressed abstractly ("specifications") or concretely ("code"). There are broadly two sorts of methods which may be used for analysis.

- Theorem proving methods are either logic-based (e.g. proving safety and liveness properties expressed in temporal logic) or algebraic (e.g. proving bisimulations in process algebra). TLA [Lam94] is an example of such a logic-based system, and CSP [Hoa85], CCS [Mil89], and LOTOS [ISO88] are examples of algebraic systems for specification and analysis.

- Finite-state analysis methods represent each component of a system as a finite-state process, and from that devise a representation of a single process describing the whole system. These sorts of methods are found most frequently in the analysis of communication protocols [Hol91, CK91, LS92b, BP93]. These methods are usually based on some notion of communicating finite-state machines [BZ83], but there are other methods such as temporal logic model checking [CES86, BCM90] and Ordered Binary-Decision Diagrams [Bry92].

Theorem-proving methods are intended to handle general behavior exhibited by all kinds of interacting systems. This generality is an expressive strength, but obtained at a cost of making verification and validation complex, usually needing considerable human resources to guide the theorem-prover. They are labor-intensive methods, not yet suited to automatic application.

Systems of processes in general and channels between them often have potentially unboundedly many states. In these cases, finite-state methods are not in general applicable unless approximations can be made. This expressive restriction is a disadvantage for finite-state methods. However, they can be used for much communications protocol and hardware design. They are very efficient with regard to human resources, since verification and validation may be accomplished automatically. The major limitation is handling the large number of global states typically generated by a real application. The number of global states that can be handled currently is roughly 10^{14} [CD92], but see also [BCM90].

In this work, we discuss a series of finite-state techniques for specifications or code involving communicating processes. These techniques are based on a particular algorithm and its variants for analyzing the communications between the processes, and they can be used for deadlock analysis, sequentialization, and reachability analysis. Some applications are introduced in the next section.

Deadlock. Two or more processes may *deadlock* if they are all waiting for events (communication events) that will not happen. The most trivial case is when two processes are each waiting for the other to send a message. Deadlock detection is normally thought of as an operation performed on a running system, due to the staggering complexity of trying to predict run-time behavior at compile-time. However, static analysis for deadlock-causing code or specification configurations is also possible, and preferable, in some cases. There are heuristic algorithms for general specifications (e.g. [Apt83, Mer90]), but if one needs exact analysis, the type of specification constructs considered must be limited.

Sequentialization. It is sometimes useful to transform code or specifications for multiple processes into code for a single process with similar behavior. We refer to such an operation as *sequentialization*. Everyday operating system schedulers perform run-time sequentialization, scheduling code from multiple operating system and user application processes to run on a single CPU. We analyze sequentialization at compile-time, rather than at run-time. Compile-time sequentialization is useful for real-time and safety-critical systems, in which the indeterminacy introduced by dynamic scheduling is often intolerable.

Reachability. A piece of code or a system state is *reachable* if some run of the system can execute that code or reach that state, starting from the normal start state. Code that is not reachable is called *dead code*. States that are not reachable are called *unreachable states*. Dead code which is discovered may be eliminated from the system code. Knowing which states, or which code, is reachable can simplify the process of verification, since fewer system execution alternatives need to be analysed.

1.2 Applications

Compilation and Debugging for Parallel Machines. Three major steps in compilation for parallel machines are *data distribution* (data-to-processor-memory mapping), *partitioning* (process-to-processor mapping), and *sequentialization* (scheduling a partition on its processor). Data distribution in the Fortran D System [HKK$^+$91, HKT91, HKT94] involves the introduction of communications that were not in the original parallel program. The Fortran 90D/HPF compiler also generates and optimizes communication between processors [BCF$^+$94]. In some cases one must analyse for deadlock after these transformations are accomplished. Algorithms for partitioning processes among processors are considered in [Sar89, YG92, GY92]. The processes in each partition have then to be scheduled on the processor to which they are assigned, i.e. they must be sequentialized. This may be done statically (i.e. at compile time), in order to plan the coherence of data flow between processors and minimize waiting time. Reachability analysis is a method of analyzing and debugging parallel programs [HM91]. Since the number of potential program states may be infinite, techniques such as *folding* have been used to reduce the states that need to be checked to a finite number [YT88, HM91]. The technique of Karp and Miller is another method of reducing a potentially infinite collection of states to a finite state graph [Bv91, KM69].

Analysis of Communication Protocols. Deadlock is a property which communication protocols must particularly try to avoid. Pre-run-time deadlock detection tests can be of considerable benefit, particularly since comprehensive testing of systems is often infeasible. Since the various components of a protocol are run on different pieces of hardware, it may be initially surprising that sequentialization is used in communication protocol analysis. Sequentialization occurs during conformance testing, in which traces of a protocol execution are generated by analyzing signal data obtained at points of control and observation [ISO91]. Traces from the tests are compared in various ways with the program specification. In order to do this, parts of the specification must be sequentialized in some manner. Reachability analysis is a well-known problem in finite-state methods and protocol analysis, e.g. [RS88, CK91, LS92b, BP93]. We have mentioned the state-explosion difficulties. Much work is devoted to devising efficient representations of the set of reachable states [Hol88, GW92, BFH91, JK91]. Finite state methods may also be used for analysis of systems with possibly infinite state spaces, e.g. [Bv91, KM69].

Software Development using CASE. Many CASE tools, especially those based on SASD or Real-Time Structured Analysis [HP88], use *Data Flow Diagrams* to specify virtual processes interchanging data. Data Flow Diagrams also occur in JSD [Jac86]. Associated with these diagrams are pseudocode specifications defining the operation of each virtual process in greater or lesser detail. Design engineers must obtain from the data-flow diagrams a *design*, which is a more or less general specification of the system that will be run on the actual hardware. For each processor in the hardware, the design includes a specification of the serial code, analysed to the level of subroutines, with specifications of the function of each subroutine. In the design step, the specifications of the many virtual processes in the Data-Flow Diagram must be converted into the far fewer actual processes to be coded and run on the target hardware. Sequentialization is thus a major step in design. The technical issues that arise in the design step are very similar to those in compilation for parallel machines, though at a different level of abstraction.

1.3 The Message Flow Graph

1.3.1 What is a Message Flow Graph?

The Message Flow Graph is an attempt to structure the control flow through the interprocess communication. We refer throughout to a communication action as a *commact*, and a communication statement or instruction as a *commstat*. So, process code *contains* commstats, and processes *execute* commacts. A message flow graph in its simplest form is an expansion of the code defining each individual process such that (the *unique-commstat condition*) every commstat in a process synchronises with *a unique* statement in another process, no matter when or how often it is executed. If all commacts involve only two processes, the message flow graph connects two such commstats together with an edge. For commacts involving more than two processes, the generalised message flow graph connects the commstats in an *edge-star* (see Chapter 8). In general, due to branching in code, a message flow graph of this simplest form will not exist - there will be a forest of such graphs, or some other more efficient representation. We deal with the most general case in which such a unique (up to isomorphism) simple connected message flow graph can be shown to exist.

For a system of communicating processes, almost everything you want to know about the communication can be determined from the message flow graph. It is a finite representation of a partial ordering on the communication actions of the processes. Partial orderings determine the execution traces of the system in the following way: any valid execution trace is a linear extension of one of the orders, and vice versa. We show how to compute the "smallest" message flow graph for the case in which there is a *single* partial ordering that determines the valid traces in the manner described above. For most communicating systems, even those for which the control structure is finite-state, there will be many such partial orderings. For example, if a process `A` contains a conditional one of whose branches communicates once with process `B` and the other once with process `C`, then the traces will be described by one partial order containing the `B`-communication and a different one, similar to the first but containing the `C`-communication at the same place in the order. Each such conditional in the system which entails a different sequence of communications along each branch will engender different partial orders in this way (similar things can happen with `while`-loops). There may be exponentially many partial orders needed to describe the traces of a system. Our work here focuses on just one such partial order at a time.

1.3.2 Loop Processes

We construct the minimum message flow graph in time linear in the number of commstats in this minimum graph for systems of *loop processes*. Loop processes, which contain both communication and computation statements, are characterized by their communication statements. We define a block of code to satisfy the *linear condition* if, when computation statements are ignored, the communication statements may be written as straight-line code without conditionals. When this rewrite is performed, we say the code is *commstat-reduced*. A simple example of code satisfying the linear condition is given in Figure 1.1. The example contains code that branches, but each branch contains the same sequence of communications with other processes, even though the computation performed with these values is different. Thus, all executions of the code perform the *same sequence* of communications, no matter which branch is followed, shown in Figure 1.2. The property, that all execution sequences perform the same sequence of communications, is the crucial property we require for loop processes.

If the body of a finite loop satisfies the linear condition and if the it-

$\{\ y \leftarrow (x+1);$
$\quad w \leftarrow$ receive(B);
$\quad$ if $P(x)$ then $\{$ send(x,C);
$\qquad\qquad\qquad z \leftarrow$ receive(B); $\}$
$\qquad\qquad$ else $\{$ send($x+1$, C);
$\qquad\qquad\qquad y \leftarrow y+1;$
$\qquad\qquad\qquad w \leftarrow$ receive(B) $+1$ $\}$ $\}$;
...
...

Figure 1.1: Code Satisfying the Linear Condition

```
receive-from-B
send-to-C
receive-from-B
...
...
```

Figure 1.2: The Communication Sequence in each Execution

eration count of the loop is known at compile time, we say that the loop is a *compiler deterministic* loop. A *loop process* is defined to be a process which consists of non-looping code that satisfies the linear condition, possibly interleaved with compiler-deterministic loops, followed by a single finite or infinite loop whose body also satisfies the linear condition and can contain interleaved compiler-deterministic loops. Code segments that consist of only computation instructions can contain internal conditionals and finite loops whose iteration count is not known at compile time. As shown in Figures 1.1 and 1.2, we can also have conditionals containing communication statements, so long as the same communication statements appear in exactly the same order in both branches of the conditional, and hence can be represented as straight-line code when the computation statements are ignored.

We assume that the code for the infinite loop is of minimum size: that is, there is no smaller infinite loop that is semantically equivalent. (If a pattern of commstats repeats, the computation code must distinguish the repeating commstats.) In essence, a loop process is a process in which all compile-time communication nondeterminism has been eliminated. The only limitation on the computational portion of a loop process is that the same communication statements always execute, independent of the outcome of the computation. (Even this limitation can be waived in certain cases, such as a numeric computation that iterates until the error term is sufficiently small).

Loop processes may be found in real-time code and scientific numerical code. For straight-line code including conditionals or some kinds of `while`-loops, determining if there is a deadlocking, respectively a non-deadlocking, execution trace is NP-hard (Chapter 9). This is true even when the code includes *just* conditionals, or *just* finite `while`-loops, and communication primitives. Code involving these constructs must nevertheless be analyzed, and for some special cases our algorithm can be used along with a method of keeping track of branches to provide precise compile time analysis. For more general cases, our algorithm can be used as a debugging tool.

We first construct the message flow graph for loop processes that use 2-process rendezvous type communication primitives and prove our construction algorithm correct. Such primitives are used in Occam, or the procedural version of CSP [Hoa78], and other primitives may often be constructed from or analyzed in terms of these primitives. We discuss how to handle asynchronous and partially asynchronous primitives in Chapter 7. The construction algorithm is extended to handle processes with n-way synchronous primitives in Chapter 8.

1.4 Background Definitions and Concepts

1.4.1 Interleaved Execution

We consider an execution of the system to be an interleaving of atomic actions of processes; such an interleaving is called a *trace*. More precisely, a trace is a sequence of interleaved atomic actions of the processes that respects the ordering of the actions within each process, and respects the semantics of the communication constructs. A system itself corresponds to the set of all its possible executions, thus to a *set of traces* (namely, the set of its execution traces).

Interleaving models are used in other contexts similar to ours [Hoa85, Lam90, Lam94], they underlie the semantics of CSP and therefore Occam, and are the outcome of the JSD methodology [Jac86]. In contrast, a partial order semantics would restrict us from making free use of other closely related work, and add an unnecessary layer of complexity to our analysis. We thus consider interleaving semantics appropriate for our task.

1.4.2 Communication Actions and Statements

A *communication action* (commact) is an action engaged in by more than one process and a *communication statement* (commstat) is a piece of source code within a process saying that the process should engage in a particular communication action at this point. The distinction between the two is important, since our algorithm operates on statements to provide information about the actions. The *communication graph* (commgraph) of a system of communicating processes S is a graph whose nodes are labelled with the process names, and where an edge joins nodes labelled `A` and `B` precisely when there is a commstat specifying communication between `A` and `B` in some process in S. The commgraph is a syntactically defined object, depending on process source code, and not on whether communication between `A` and `B` actually occurs during any execution of S.

1.4.3 A Minimum System Loop Process

Because commacts are atomic, and traces are interleaved commacts, we can describe a non-deadlocking trace of a set of communicating loop processes as a single loop process, which we call the *system loop process*. The system loop process consists of a *system preamble* followed by a *system* loop body. A commstat in the system loop process represents a single commact. We define a *minimum* system loop process to be a system loop process which contains the minimum number of commstats of any system loop process that corresponds to the same trace. We say that a message flow graph is *minimum* if it contains the minimum number of nodes of any message flow graph for that system of processes. In Chapter 6 we show that the message flow graph we construct is minimum, and that it corresponds to a minimum system loop process. (We give there a rigorous definition of the notion of 'corresponds'.)

1.4.4 Communication Primitives and Events

Our results and our algorithm are very sensitive to the exact choice of communication primitives. A communication primitive is:

- *synchronous* if it denotes a single indivisible action which is partaken of by multiple processes;
- *blocking* if a process that is ready to communicate must suspend until all other participating processes are ready to communicate;
- *n-process* (*two-process*) if every communication involves at least two processes, and no more than n (two) processes are involved in any communication;
- *transparent* if the communications can be modelled as a k-way (two-way) channel between the processes over which communication takes place, where $k \leq n$ is the number of processes participating in the communication; (i.e. each process involved in a communication event knows the names of all other processes involved in that event);
- *acceptable* if it is synchronous, blocking, and transparent;
- *convenient* if it is acceptable and two-process.

For simplicity in our examples, an event is always a `Write` from some process `A` to some process `B`, which we denote by the statement `W(AB)` in both processes. Thus, `W(AB)` is a statement corresponding both to a `Write-to-B` in `A` and a `Read-from-A` in `B`. The communication succeeds if and only if `W(AB)` appears in the appropriate place in both processes, and both statements are executed. With slight abuse of notation, we refer to `W(AB)` also as the *event type* of the action (event) it denotes. (Our formulation is different from but equivalent to that used in CSP and Occam).

1.4.5 Deadlock Analysis and Sequentialization

A finite representation of an execution trace is called a *sequentialization.* A *deadlock-free sequentialization* is a sequentialization of a trace that does not deadlock. A *deadlocking sequentialization* is a sequentialization of a trace that deadlocks. A deadlocking sequentialization cannot include code for all source commstats, since some are not executed, by definition of deadlock. It includes a representation of all commstats up to the deadlock condition. Our algorithm performs deadlock analysis and sequentialization of code for loop processes at compile time. If there is deadlock, the version of the algorithm we present halts when deadlock is discovered; it may easily be modified to produce a maximal trace up to the deadlock condition, or to continue analyzing remaining processes not involved in the deadlocked cycle.

We represent traces by minimum system loop processes; a sequentialization is isomorphic to serial code, the execution of which is consistent with the process definitions. The analysis for loop processes is made easier because either every trace deadlocks, or every trace is deadlock-free. A system of communicating processes with this property we define to be *uniform.* For a uniform system, there is no distinction between *angelic* sequentialization (if any trace is deadlock-free, such a sequentialization is generated) and *demonic* sequentialization (only if all traces are deadlock-free is a deadlock-free sequentialization guaranteed to be generated) [Sch90]. (See corollary 6.3.4). The form of the sequentialized code for loop processes is the same as that of each individual loop process, namely `(A; loop B endloop)`. (Infinite or potentially infinite loops are generated by the `loop...endloop` construct, defined in Chapter 2). Figure 1.3 gives an example of straight-line code (loop processes) using Ada rendezvous which is not uniform.

1.5 Some Difficult Questions

1.5.1 What About Analyzing Ada?

Ada rendezvous is a different sort of operation from those we consider here. An acceptable primitive is a rendezvous-type communication, but, contrary to the comment in [Apt83], it must be distinguished from Ada rendezvous, which is non-transparent. Unlike loop processes, straight-line code using the Ada rendezvous primitive can have some traces that deadlock and others that don't. Figures 1.3 and 1.4 give non-deadlocking, respectively deadlocking, traces of the same Ada code constructed using the rendezvous primitive and

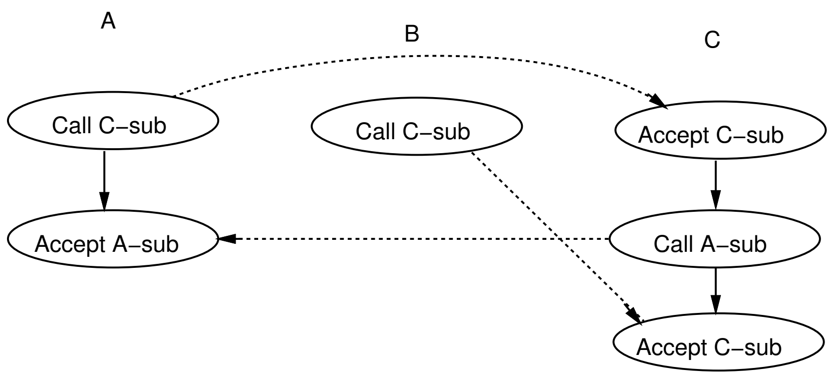


Figure 1.3: A Non-Uniform Ada-like Example: A non-deadlocking trace

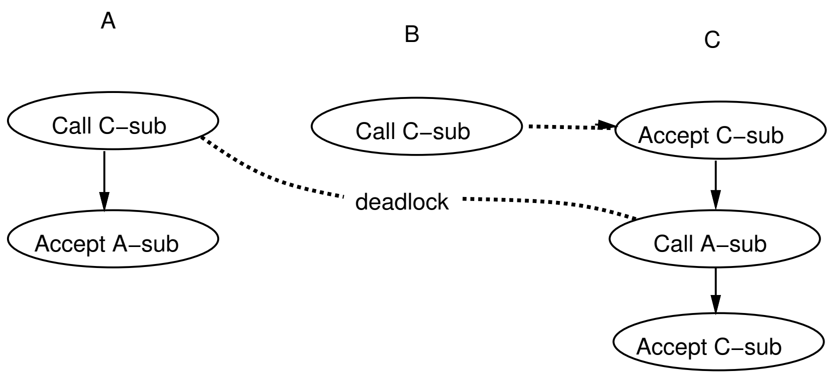


Figure 1.4: A Non-Uniform Ada-like Example: A deadlocking trace

sequential composition only.

1.5.2 How Hard is Everything?

Almost everything is NP-hard or exponential. However, determining whether there is a trace that deadlocks is in NP in all the cases we talk about, whether just for loop processes, or also code including conditionals and finite loops.

Determining that there is a trace that does not deadlock is NP-complete for a system of processes with straight line code, three or more conditionals, and no loops of any sort. A similar result holds for processes with no conditionals, no infinite loops and only three finite loops whose iteration number is either 0 or 1, so long as the exact value of the loop iteration number cannot be determined at compile time. Determining if there is a trace that *does* deadlock is also NP-complete if "three" is replaced by "six" in the above two examples. By contrast, our algorithm runs in time linear in the size of the input for loop processes satisfying the linear condition with no conditionals or infinite loops, so long as the finite loop iteration numbers are constant and known at compile time.

In addition, we have shown that determining that there is a trace that does not deadlock is NP-complete for a system containing only two processes and no conditionals or loops, if the dependence graph of one of the processes consists of only a set of chains (threads) and the other dependence graph is only straightline code (a single chain or thread) (Chapter 9). This result suggests that analysis is going to be difficult for processes with even slightly complex dependence graphs. We have proven that the problem of determining if a loop process satisfying the linear condition deadlocks is in NP, but we do not know if the problem is NP-complete. Our algorithm for loop processes may produce output necessarily exponential in the size of the input, but linear in the size of the minimum system loop process. All these results may be found in Chapter 9.

1.5.3 Aren't Loop Processes Somewhat Limited?

Loop processes are a limited program construct. When computation statements are ignored, they contain no conditionals, no indeterminate finite loops, no execution indeterminism of any sort.

The communication graph for our encoding of *One-in-Three 3-SAT* in one of our NP-completeness reductions in Chapter 9 is a fairly simple tree. By comparison, for systems of unrestricted loop processes with a communication graph that is a tree, deadlock analysis is in P [Sim]. This provides evidence that loop processes are conceptually simpler to analyse even than code involving simple conditionals. The result in [Sim] also shows that it is not necessary to build the message flow graph to determine deadlock, since our example of an exponentially large message flow graph in Chapter 10 has a tree-like communications graph. The only successful approaches to handling conditionals and other non-deterministic constructs are heuristic [Apt83, Mer90]. It is fairly easy to see that construction of the message flow graph for branching processes, where different branches of a process contain commstats to different processes, necessitates a corresponding branching of the message flow graph. Different continuations of the message flow graph are generated by the traces along different branches, when the communication partners with each branch are different, or when communications with different partners occur in a different order.

In cases in which processes may branch, but this branching does not affect the sequence of commacts taken along each branch, there is no need to branch in the construction of the flow graph.

Because of the potential on an exponential explosion in branching, we have limited our analysis in this monograph to the case in which branching is not required for the message flow graph. We observe that the Sequence Condition (Chapter 5) can be a useful tool in the construction of heuristics for the general case.

1.6 Some Other Relevant Work

Much work has been done in a variety of contexts on deadlock analysis and sequentialization. Apt [Apt83] and Mercouroff [Mer90] analyze general CSP processes. CSP is a process algebra employing synchronous primitives, able to specify general processes. The deadlock analysis problem is noncomputable in the general case, and so they provide heuristic algorithms.

Sridhar and Hoare [SH89] consider an example of multiple CSP processes which communicate via process calls, i.e. a system of mutually recursive processes. They are interested in sequentializing these processes as a way to eliminate the recursion. The main point of their paper is to show how the JSD development methodology can be carried out within CSP, and as we note in Section 1.7.8, sequentialization is an important component of the JSD methodology.

Taylor gives an algorithm to perform deadlock analysis of Ada tasking

[Tay83]. His algorithm is limited to the case of a tree-like communications graph, and is exponential. [Sim] shows that the deadlock analysis problem is polynomial for synchronous transparent primitives with a non-cyclic communications graph. Our demonstration of exponential blow-up in construction of a message flow graph uses a linearly ordered communications graph.

Reif and Smolka [RS90] consider communication message flow analysis for asynchronous communication. Some of our results in Chapter 9 improve on some of theirs.

Cuny and Snyder [CS87] convert systems of processes using asynchronous communication into systems which communicate synchronously. Their interest is in compilation of scientific and numerical code for parallel machines. In Chapter 5 we give an explicit condition, the Numerical Matching Condition, for determining when systems are *balanced* in the sense of [CS87]. This explicit condition is used in proving termination of the message flow graph construction algorithm.

Schneider's thrifty execution method [SCS85] may also be adapted for sequentialization of code in which the dataflow forms a tree [Sch90].

A formal semantics for message flow graphs, interpreted as global state graphs and then as ω-automata, is given in [LL94a]. (This semantics has been used to interpret the telecommunications description devices known as Message Sequence Charts [LL94b].) A finite-state automaton of global states (e.g. a Büchi automaton in [LL94b], but see [LL95] for a discussion) is derived from a message flow graph, and it is noted that the number of global states could be exponential in the size of the message flow graph. The set of global states of a system of loop processes may thus be derived from the message flow graph, and the number of global states may be exponential in the size of the message flow graph.

Thus the message flow graph can be considered as an efficient representation of the set of global states of the loop process system. These considerations link the message flow graph construction to a collection of other attempts to provide efficient representation of the set of global states of a finite-state system. We believe it is closest to the representation in [GW92], but see also [BFH91, BFH$^+$92, JK91].

Other efficient ways of representing traces of systems of communicating processes may be found using Trace Theory [Maz87], which is based on partially-commutative monoids. Extensive work on traces in this sense may be found in [Die90a] and [Die90b].

A characterisation of classes of systems of communicating finite-state machines (CFSMs) with decidable deadlock detection problems is given in [PP91]. Algorithms for deadlock detection and reachability for CFSMs whose communication graph is a simple cycle are given in [PP92]. (This should also be compared with [Tay83], which handles tree-like communication.) In contrast, our algorithm handles simpler processes, but arbitrary communication graphs.

1.7 Some Particular Applications

In Section 1.2, we gave three application areas in which we feel our work could be useful. In this section, we briefly point out specific applications of the work in these areas.

1.7.1 Debugging Traces

Our algorithm does not explicitly handle processes that contain compile-time nondeterminism, such as conditionals or finite loops whose iteration count is unknown at compile time. Nonetheless, it can be used as an effective debugging tool. One approach for testing the code is for the programmer to specify the values of the conditionals and the number of loop body iterations. The algorithm can then determine whether or not such an execution will deadlock. At the other extreme, the code may be run with its full nondeterminism; if there is deadlock, our algorithm utilizes information about the run time choices to analyze the deadlock.

1.7.2 Analysis of Code for Transputer Networks

The Occam language, used for programming transputer networks, relies on a convenient communication primitive [Jon87]. A typical configuration incorporates multiple communicating transputers, using Occam as the assembly language. Our algorithm can be used to detect deadlock at compile time in Occam loop-process code for transputer networks. One can test such systems by checking actual execution traces against the message flow graph derived from a specification, rather than having somehow to check against the specification directly.

1.7.3 Analysis of Real-time Code

Real-time code must not deadlock, otherwise real time constraints (deadlines) cannot be satisfied. Therefore, deadlock analysis must be performed, if at all, at compile time. All correct sequentializations, and all correct traces, can be generated from the message flow graph. It is straightforward to determine if deadlines are met in a particular sequentialization, given timing values for the individual instructions. If all deadlines are met by a particular sequentialization, this can be used to implement the system.

1.7.4 Sequentialization of Code

Software systems are frequently designed using a variant of Structured Analysis/Structured Design or its real time versions, presented in [WM86, HP88]. The requirements specifications have the form of a collection of many communicating processes, but typically the code design is targeted for a limited number of processors. Consequently, specifications must be sequentialized in the design stage, which is currently accomplished by hand. Our sequentialization algorithm can be used to help automate this process.

The algorithms discussed by Simons, Ferrante and Sarkar for sequentializing code containing the parallel composition operator [SF93, Sar92, SS94] could possibly be integrated with ours to provide a sequentialization capability for a wide variety of parallel programming languages.

1.7.5 Analyzing CSP Processes

CSP is a well-known system specification and design language [Hoa85], [SH89]. The deterministic version of CSP uses a convenient communication primitive. It follows from our analysis in Chapter 9 that deadlock detection for loop-free specifications that include guarded communication is NP-hard. (By *guarded communication* we mean statements of the form `(if` θ `then Read/Write(P))`, where θ is some predicate). For loop processes without non-deterministic choice or guarded communication, the message flow graph constructed by our algorithm may be used to interleave code for CSP processes which communicate in arbitrary ways.

1.7.6 Analyzing LOTOS Processes

LOTOS [ISO88] is an ISO standard specification language which incorporates aspects of CSP and CCS [Mil89]. The communication primitive of LOTOS is a multiway synchronous communication, and our algorithm in Chapter 8 may be used to analyze LOTOS processes similarly to the way in which CSP processes may be analyzed using the 2-process algorithm.

1.7.7 Compilation of Parallel Code

Parallel systems are being designed using message passing as the basis of communication. The asynchronous version of our algorithm (Chapter 7) can be used for debugging or compiling code written for some message passing systems.

Cuny and Snyder define loop processes using a definition that is similar but not identical to ours, and show how systems of loop processes using asynchronous, buffered, two-process, transparent communication can be transformed, under certain circumstances, into systems of loop processes using a convenient communication primitive [CS87]. An example they give is numeric code for conjugate gradient methods for 2-dimensional elliptical PDEs, which is written in terms of (their) loop processes. Our techniques may be used to analyze the output of the Cuny/Snyder algorithm.

1.7.8 Generalized Program Inversion in JSD

The Jackson System Development method is used widely for software development [Jac86]. *System Specification Diagrams* specify *inter alia* the *data streams* between processes. Data streams are unidirectional synchronous or asynchronous communications of structured data between two specified processes. Various formal and informal transformations are used to convert the specification into a design for software in a procedural language. One step converts a specification of processes connected via data streams (a data transformation specification) into a design for a single process. A technique called 'program inversion' is recommended [Jac86, Chapter 11] for sequentializing processes with restricted (linearly ordered) data flow graphs. Our method can be used to sequentialize loop processes (a more restricted type of process) with arbitrary data flow graphs. We call these data flow graphs *communication graphs*.

1.8 Overview of the Following Chapters

Chapter 2 defines some terms from compiler optimization theory. Chapter 3 discusses the message flow graph and presents the algorithm for generating it, with an example. Chapter 4 presents the termination and correctness proof, along with a proof of uniformity. Chapter 5 presents the Sequence Condition and the Numerical Matching Condition (NMC). The NMC is used essentially in the proof of termination in Chapter 4. Chapter 6 discusses traces, sequentialization, and proves that the message flow graph we construct is minimal. Chapter 7 modifies the algorithm to handle asynchronous and partially-asynchronous primitives, and Chapter 8 modifies the algorithm for multiway communication primitives. Chapter 10 addresses the complexity of various problems and of our proposed solutions.

Some of this material has been presented in part in [LS92a] (Chapter 3), [LS94b] (Chapter 5), and [LS94a] (Chapter 8).

Chapter 2

Manipulating Loops

To analyze loop processes, we perform some well-known transformations on loop code, see e.g. [ASU86]. The `unroll` and `duplicate` transformations are behaviour-preserving on infinite loops, but not on finite loops without some modification, and some restriction on `duplicate`. This short chapter introduces these transformations in detail. Pseudocode for `unroll` and `duplicate` may be found in Figure 3.3.

The `loop...endloop` Construct. The `loop...endloop` construct specifies an infinite loop iteration in which the body of the loop is executed without termination criteria. We can obtain the usual `while`, `for`, and `repeat` loops using this construct, while maintaining invariance under certain program transformations that we need, namely unrolling and duplication.

For example, a `while` loop can be constructed by including within its body at the appropriate point a *conditional-exit* statement of the form `if` θ `then exit`, for some predicate θ. When θ becomes true, the next execution of this statement will cause control to pass out of the loop, which is exactly the behaviour required of a `while` loop.

Our *unrolling* transformation (below) performs a loop *rotation.* A rotation is just a cyclic permutation of the loop body, for example the rotations of `(loop a;b;c endloop)` are `(loop b;c;a endloop)` and `(loop c;a;b endloop)`. In order that unrolling preserves the semantics of the loop, it is necessary that the loop exits at the same point after unrolling as before.

Consider a `while` loop `(while` θ `do a;b;c endwhile)`, and a rotation `(while` θ `do b;c;a endwhile)`. In the rotation, the loop predicate θ is tested after execution of `a` and before execution of `b`, whereas in the original loop θ is tested after `c` and before `a`. Thus the original loop exits directly after some execution of `c`, whereas the rotation exits directly after some execution of `a`; hence the loop and the rotated loop do not exit at the same point in the program. Such a rotation would be forbidden in our model.

In contrast, consider the equivalent formulation using our construct, `(loop a;b;c;(if` θ `then exit) endloop)`. A rotation is `(loop b;c;(if` θ `then exit);a endloop)` and the exit test appears in the same relative position in the code. Thus rotation in this construction does not change the position where the loop exit predicate is tested during execution. (In both constructions, the loop entry is changed. This is explicitly handled in the *unroll* construction so that the same semantics is maintained.)

We consider n loop processes $\mathtt{P}_1 \ldots . \mathtt{P}_n$, where $\mathtt{P}_k$ has the form $(\mathtt{A}_k;$ `loop` $\mathtt{B}_k$ `endloop)` for $k = 1, .., n$, and the commstats in `A` and `B` form blocks of straight-line code (no loops, no conditionals, possibly some communications). For the purpose of analysis, we ignore all computation statements, upon which no restrictions are placed by our model. We refer to $\mathtt{A}_k$ as $\mathtt{P}_k$'s *preamble code* or $\mathtt{preamble(P}_k\mathtt{)}$, and to $\mathtt{B}_k$ as $\mathtt{P}_k$'s *loopbody code* or $\mathtt{loopbody(P}_k\mathtt{)}$. We assume that any finite loop whose iteration number is known at compile time has been unrolled and incorporated into the preamble or the loop body, and that $\mathtt{loopbody(P}_k\mathtt{)}$ is of minimum size.

Unrolling and Duplication. Since all our analysis involves only commstats, we assume that any other instructions in $\mathcal{L}$ are unrolled in a consistent fashion. Suppose process `P` consists of the commstats $(a_1; \ldots; a_k$ `loop` $b_1; \ldots; b_j$ `endloop)`. The *unrolling* of `P` *down to* b_k is:

$$a_1; \ldots; a_k; b_1; \ldots; b_k \;\mathtt{loop}\; b_{k+1}; \ldots; b_j; b_1; \ldots; b_k \;\mathtt{endloop}$$

Since the `loop...endloop` construct specifies an infinite iteration of the loop, as we execute an unrolling of a loop, we are executing the same sequence of statements as in the original loop. The *duplication* of a loop body is the concatenation of the body to itself an integral number of times. Specifically, the *m-duplication* of $\mathcal{L}$ is the loop obtained by m consecutive repetitions of the body of $\mathcal{L}$ inside the `loop ... endloop`. A *duplication* of $\mathcal{L}$ is any m-duplication. We define *m-duplication* of $\mathcal{L}$ for integers $m \geq 1$ by induction. The *1-duplication* of $\mathcal{L}$ is $\mathcal{L}$. Let $\mathcal{M}$ be the body of the m-duplication of $\mathcal{L}$. Then the *$(m+1)$-duplication* of $\mathcal{L}$ is `loop` $i_0; i_1; \ldots; i_{k-1}; \mathcal{M}$ `endloop`.

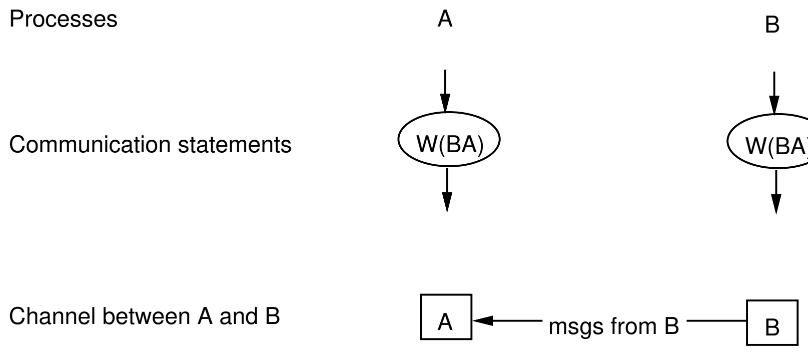


Figure 2.1: Communicating processes

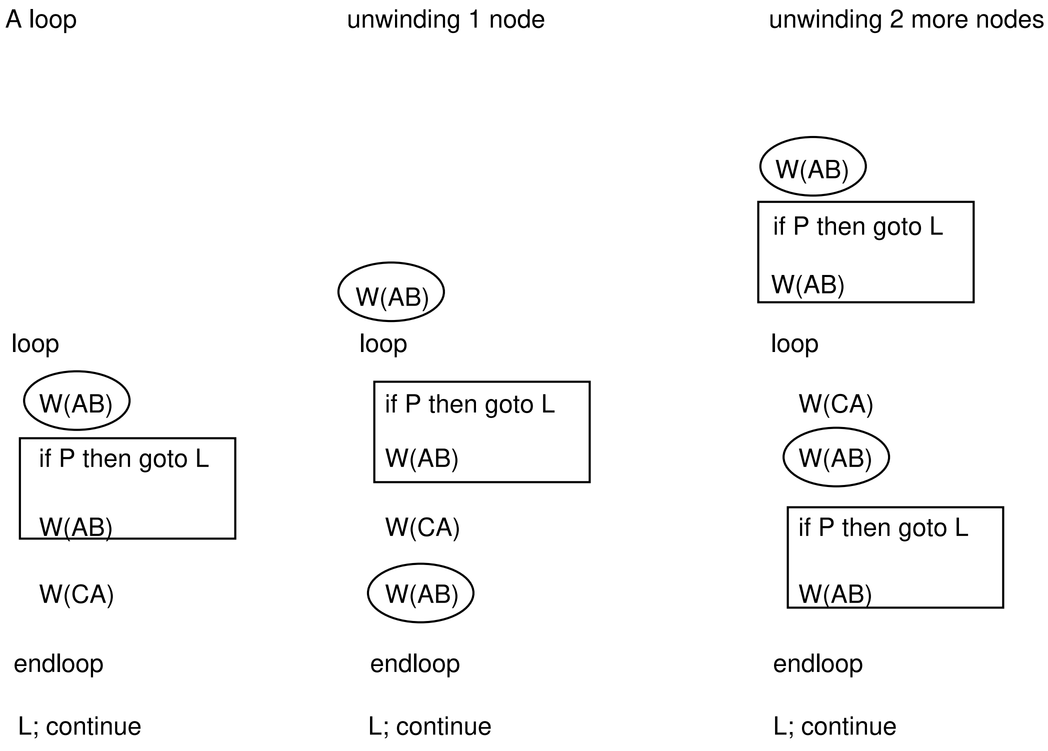


Figure 2.2: Loop unrolling

Matching Loopbodies. Let β_i and β_j be two loopbodies, and let b_{ij} (b_{ji}) be the number of commstats in β_i (β_j) involving β_j (β_i), (i.e. the number of `W(`$\mathtt{P}_i$`,`$\mathtt{P}_j$`)` in each). β_i and β_j *nummatch* if $b_{ij} = b_{ji}$; β_i and β_j *match* if β_i and β_j nummatch and all commstats involving $\mathtt{P}_i$ and $\mathtt{P}_j$ can be matched (a definition of *matched pair* of commstats is given later in Section 3.1.3). A set of n loopbodies nummatch if every pair of loopbodies β_i and β_j nummatch for $i \neq j; i, j \leq n$. If a set of loopbodies can be made to nummatch using duplication, we say the original loopbodies are *nummatchable*. (This is the same notion as that of *balanced* in [CS87]).

Chapter 3

The Message Flow Graph Algorithm

3.1 Introduction

3.1.1 What the Algorithm Does

Our algorithm constructs the minimum message flow graph (*mfg*) for loop processes. (The term "message flow graph" is defined formally in Section 3.1.3. Minimality of the flow graph we construct is shown in Chapter 6.) The mfg represents commstats in the process code as nodes. Two consecutive nodes (commstats) in an individual process are connected by a directed edge (D edge), and two nodes representing the same communication action (commact) in two different processes are connected by an undirected edge (U edge). We represent the infinite sequence of commacts specified by the infinite loops by the `loop ... endloop` construct. The valid execution sequences of the loop body consist of an infinite concatenation of finite sequences of commacts. Each finite sequence represents a sequence of actions 'once through the loop', and is a linear extension of the partial ordering denoted by the graph for the loop body.

3.1.2 Discovering Deadlock Conditions

The interprocess undirected U edges in the mfg represent synchronization points (the occurrence of a commact at this point in both processes), and the intraprocess directed D edges represent the ordering of code inside a process, by connecting one commstat to its immediate successor. (This

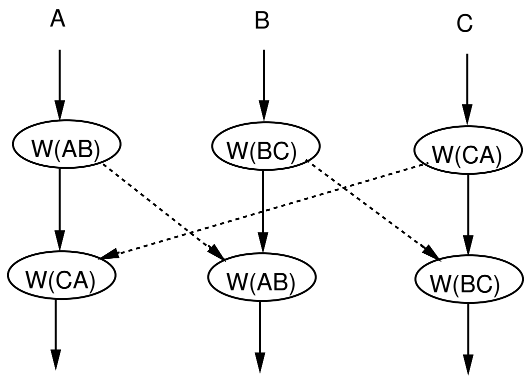


Figure 3.1: A Cycle

successor is unique for loop processes, so the ordering of code is linear, and also transitively reduced.)

A *cycle* is a path that begins and ends at the same node, that traverses each D edge respecting the direction of the edge, and may traverse a U edge in either direction. A cycle represents a deadlocked state, since it implies that certain intraprocess statements (D edges) are executed, along with certain communication actions (represented by U edges), and yet we end up back at exactly the same point in a trace, which is impossible. We state this more formally in Chapter 4. Figure 3.1 shows a cycle. The *Cycle Condition* is the condition that the message flow graph has no cycles.

The *Isolated Node Condition* states that each node has a matching node with which it participates in a commact, that is for every `W(AB)` in `A` there is either some statement `W(AB)` in `B` that is not already assigned to a different commact, or such a node will appear as the loop body is duplicated. Jointly, the Isolated Node Condition and the Cycle Condition provide necessary and sufficient conditions for there to be no deadlock, as we prove more formally in Chapters 4 and 5.

The *Numerical Matching Condition* (NMC) and the *Sequence Condition* (SC) are necessary conditions for there to be no deadlock. (If either is violated, there will be either a cycle or a node that can not be matched). The NMC is also needed to show termination of the algorithm.

If there is no deadlock, the algorithm can produce a sequentialization along with the message flow graph. If there is deadlock, the version of the algorithm that we present halts with a notification as soon as deadlock is found. However, one may wish to continue the analysis for those processes that are not deadlocked; the algorithm may be easily modified to continue for processes not involved in the deadlock. In this case, both nodes in deadlocked processes which are above those nodes involved in a deadlock cycle and nodes in processes not involved in the deadlock cycle may continue to be matched with their partners in communication.

3.1.3 The Message Flow Graph

Our algorithm constructs the mfg $G = (N, D, U)$, where D and U are sets of respectively directed and undirected edges, defined below. N is a set of labelled nodes, whose labels are the commstats in the system of processes P_i, $1 \leq i \leq n$. Define `label(`x`)` to be the label assigned to x. The notation $x \in \mathrm{P}_i$ signifies that (statement) x occurs in P_i; if $x \in \mathrm{P}_i$, then we refer to P_i also as $\mathrm{P}(x)$. The following definitions hold both for the original loop process P_i and for all forms of P_i in which `loopbody(P`$_i$`)` has been unrolled or duplicated. For commstats $x, y \in \mathrm{P}_i$, y is the *immediate successor* of x if y is the first commstat to occur immediately after x in the commstat-reduced code of P_i. The first statement of the loop body is the immediate successor of the last statement of the preamble. For commstats $x, y \in \mathrm{P}_i$, x is *above* y if $x, y \in \mathrm{P}_i$, and there exists $z \in \mathrm{P}_i$ such that y is an immediate successor of z and either $z = x$ or x is above z.

A D edge is a directed edge $\langle x, y\rangle$ such that y is the immediate successor of x in $\mathrm{P}(x)$. All D edges are in D. If `loopbody(P(`x`))` is unrolled or duplicated, then the algorithm places a D edge between each duplicated node and its immediate successor. We say that the pair of commstats $\{x, y\}$ is a *matched pair* if `label(`x`) = label(`y`) =` α, and all nodes with label α above x in `P(`x`)` and all nodes with label α above y in `P(`y`)` are already part of U edges. The algorithm will ultimately place U edges between elements of a matched pair if no deadlock is found. The *input graph* consists of the commstat-reduced code for each process, with D edges between each node and its immediate successor (and no U edges).

Formally, a two-process *message flow graph* (mfg) for the input graph is a graph obtained from the input graph by unrolling and loop-duplication, in which

the two-process unique-commstat condition every node is connected by a U edge to precisely one other node, and such that

the trace condition if x is an mfg node connected by a U edge to y, then in every trace in which the commact denoted by commstat x is executed, the commact denoted by x is identical with the commact denoted by y.

Note that it follows that the traces of the input graph and the mfg are identical, since the mfg may be obtained by unrolling and loop-duplication from the input graph. We use this result without proof, since it is well-known in compiler optimisation theory.

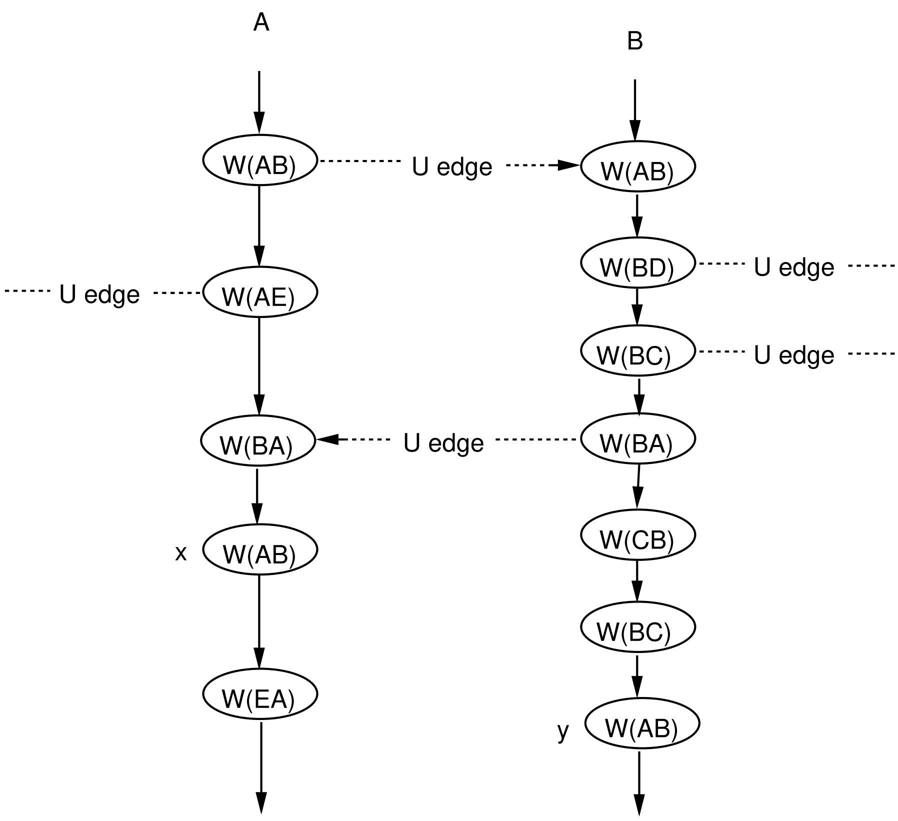
We will prove the two-process mfg algorithm correct in detail. In Chapter 8 we shall generalize the mfg construction to n-process synchronous communication primitives. We shall need there to generalize the current notion of mfg by replacing the two-process unique-commstat condition by the multiway unique-commstat condition.

Our algorithm constructs a graph by unrolling and loop-duplicating the input graph. It connects every node to precisely one other node, by Lemmas 4.1.1 and 4.1.2, and the trace condition is proved in Theorem 4.1.4.

Assume that the iteration count of all finite loops is known at compile time and that they have all been unrolled. An *isolated node* is a node that is not (yet) part of a U edge. A node is *matched* if it is part of a U edge or a matched pair. An isolated node becomes incorporated into U edges as the algorithm proceeds, unless there is no node with which it can be matched. In this case the algorithm terminates with notification of the deadlock condition, together with the name of the isolated node or the information necessary to reconstruct the deadlocking cycle. The *topmost isolated node of type* α in a process is an isolated node of type α with no isolated node of type α above it. The *topmost isolated node* (tin) in a process is an isolated node with no isolated nodes of any type above it. The nodes identified as *topmost isolated node* or as *topmost isolated node of type* α change as the algorithm proceeds and U edges are drawn.

3.1.4 A Simple Preliminary Check

Let $AB =$ the number of nodes in the preamble of process A that communicate with process B. Similarly, let $ab =$ the number of nodes in infinite loop code of process A that communicate with process B. Likewise for BA, ba. Note that if there is no infinite loop in A, $ab = 0$. The proof of the following lemma is obvious.



The D edges are omitted for clarity. The topmost isolated node in `A` is `x`, the second `W(AB)` node in `A`. `x` and `y` comprise a matched pair, but `W(CB)` is the topmost isolated node in `B`. `W(BC)` is another isolated node that is above `y` in `B`.

Figure 3.2: A Matched Pair and Isolated Nodes

Lemma 3.1.1 *Processes A and B will deadlock in either of the following two circumstances:*

(i) $ab = ba = 0$ *and* $AB \neq BA$

(ii) $ab \neq 0$ *but* $ba = 0$

3.2 The Algorithm

The message flow graph algorithm itself is presented in Figures 3.3, 3.4, and 3.5. Its operation is explained by example in Section 3.3. The algorithm is presented in sufficient detail to enable a detailed proof of correctness to be given, in Chapter 4.

We assume an implementation language with set and sequence data types. We use the symbols $\frown$ for concatenation of sequences, / for its right inverse (deleting a sequence from the tail), and $\langle\rangle$ for the null sequence. For a sequence s, we use the function `last(s)`, which returns the last element of s. For sets, we use $\cup$ for set union, and '$-$' for set difference. In the algorithm, we also use the function `input-size(P)`, which returns the number of statements in the code fragment `P`.

3.2.1 How the Algorithm Executes

Every event type is associated with two processes that participate in an event of that type. For example, the type `W(AB)` is associated with processes `A` and `B`. Given a statement x of event type α in process `P(x)`, we denote the other process involved in event type α by `processmate(`x`)`. (The algorithm is built upon the uniqueness of `processmate(`x`)`. For Ada, the lack of a unique label for a commstat, because of the lack of a unique processmate, is the primary distinguishing factor between acceptable communication primitives and Ada rendezvous).

Execution of the algorithm is similar on preamble code and on loop body code, with the exception of how code is duplicated to allow correct matching. A topmost isolated node is selected, and its match is found in the processmate. If this match is itself a topmost isolated node, the U edge is immediately drawn. If not, then the matched pair is put on the stack, and the topmost isolated node in the processmate is processed by a recursive call. This recursive process can be thought of in the following way: a U edge is drawn only if that edge represents a communication action that *could* go next in a valid trace. If not (i.e. there is an isolated node in the processmate above the matched node), then processing of that pair is suspended, and processing of an node denoting an action that *must* be earlier (namely, the *topmost* such node) is started.

In addition to popping the stack if a U edge can be drawn, the recursive calls may terminate with an exit due to either an unmatchable isolated node, or to the discovery of a cycle.

We first process all preamble nodes. If a preamble node is matched with a loop body node, then the loop body is unrolled the minimum amount required to put a copy of the loop body node in the preamble. This allows preamble nodes in the message flow graph to be connected only to other preamble nodes (and thus output loop nodes to be connected also only to output loop nodes). We formally prove this in Chapter 4. It should be noted that some preamble nodes in the message flow graph may have been in the loop bodies of the input to begin with (and copies of them still remain there).

After all preamble nodes have been processed, we process loop body nodes. If there is no unmatched node for a loop body node to match, then the subroutine `match` calls `duplicate`, which appends another copy of the original (rotated) loop body to the end of the current loop body. These two different actions mark the difference between preamble processing and loop body processing. Consequently, we use a flag `preamble` to discriminate between the two types of processing, and different actions are taken based on the value of the flag. The loop bodies are duplicated because the original loop bodies are assumed to be minimum, and consequently the loop body (possible rotated) must appear an integral number of times in any loop body that has identical semantics to the original loop body. The NMC specifies when loop bodies can be made to match through duplication.

3.2.2 Algorithm Output

Since loops may be unrolled and loop bodies may be duplicated, both the final preambles and final loop bodies may contain more code than the initial ones. Suppose the initial code of a process is $(\mathtt{A}_k\mathtt{;}\ \mathtt{loop}\ \mathtt{B}_k\ \mathtt{endloop})$. The final preamble of a single process is $(\mathtt{A}_k\mathtt{;}(\mathtt{B}_k)^*\mathtt{;}\mathtt{C}_k)$, where $(\mathtt{B}_k)^*$ is a finite number, possibly zero, of copies of $\mathtt{B}_k$, and $\mathtt{C}_k$ is an initial segment of $\mathtt{B}_k$. In this case, if the initial loop body is $\mathtt{B}_k = \mathtt{C}_k\frown\alpha$, then the rotated loop body which is input to the loop body part of the algorithm is $\alpha\frown\mathtt{C}_k$.

Suppose the algorithm is processing loop body code. If in attempting to match a node, it determines that all nodes of the appropriate type in processmate have already been matched, it must duplicate the loop body code of the processmate to provide an unmatched node of the right type. As mentioned above, the code loop body must be duplicated whole, to maintain the same semantics.

The subroutine `match` calls `duplicate` on loop body code if there is no isolated node of the right event type in the processmate, but there is a matched node of the right type in the loop body of the processmate. The final loop body is a multiple duplicate of a rotation of the original loop body, i.e. the loop body of the output will consist of one or more copies of $\alpha \frown \mathtt{C}_k$.

As we show in Theorem 4.1.3, the algorithm always terminates, having either detected deadlock or having constructed a message flow graph if desired. (We show in Chapter 6 that this message flow graph is minimal and that the algorithm constructs a sequentialization.) If deadlock is caused by a cycle, the variable `Path` contains the information necessary to construct the cycle causing deadlock (`cycle-exit`). If deadlock is caused by an isolated node that cannot be matched, even with loop unrolling or duplication, the isolated node is also identified (`isolated-node-exit`).

3.2.3 Preprocessing

In a single pass over the input, one may determine the input graph, as well as the sequence of statements corresponding to a particular event type, in time linear in the number of commstats. These structures can be represented as linked lists and used in the subroutine `first-node`. Code for these data structures is not included here.

3.2.4 The `ptin`, `handle`, and `match` routines

These three routines constitute the core of the algorithm. The routine `ptin` processes topmost isolated nodes (`ptin` is the acronym). `match` is called by `ptin` to find the node which matches an isolated node being processed, and `handle` is called to process a matched pair, and eventually turn it into a U edge if deadlock is not detected. The routines `ptin` and `handle` are mutually recursive.

3.2.5 The `unroll` and `duplicate` subroutines

The `unroll` subroutine is shown in Figure 3.3. Suppose process P is

$$(a_1; \ldots; a_k; \ \mathtt{loop}\ b_1, \ldots, b_n\ \mathtt{endloop})$$

In unrolling down to the node with label b_j, we concatenate nodes $b_1, \ldots, b_j$ to the end of `preamble(P)`. We then create new nodes (with unique names) with labels $b_1, \ldots, b_j$ to append to the end of `loopbody(P)`. We also add D edges between the last old loop node, and the first new loop node, and between each immediate-successor pair of the new loop body nodes. The function `name(x)` returns the unique name of node x, and `newnode(`b`)` returns a new node z, such that `label`$(z) = b$.

`Duplicate` is similar, and also shown in Figure 3.3. `Duplicate` adds one copy of the *original* (possibly rotated) loop body code to the end of the current (loop body) code, which contains an integral number of copies of the original loop body code.

3.2.6 The `main` routine

The `main` routine is shown in Figure 3.4. We consider a process P to be a sequence of graph nodes that represent statements, with a D edge between each node and its immediate successor. Thus, P is a subgraph of G. To execute the algorithm, `main` is called on the input graph G $= \langle$N, $D_0, \emptyset\rangle$. N is the collection of nodes corresponding to the commstat-reduced process code, and D_0 is the collection of D edges between statements and their immediate successors in the commstat-reduced process code. The predicate `preamble` is set to *true* so long as some preamble node is not matched; as soon as all the preamble nodes are matched, `main` sets `preamble` $=$ *false*. The code in `main` for both kinds of nodes is identical, but the value of `preamble` affects the processing within `ptin`, `handle`, and `match`.

3.2.7 Cycle Detection

The variable `Path` maintains a sequence of process names of nodes processed recursively by `handle` during a single call of `ptin` in `main`. At the beginning of each call to `ptin` the name of the process containing the node currently being processed by `ptin` is added to `Path`; immediately after `handle` returns, `ptin` deletes the process name from `Path`.

If the same process name appears twice in `Path`, a cycle can be constructed by including the edges between the topmost isolated nodes x in each of the processes in `Path` to their respective processmates `processmate(`x`)`, together with the D edges between `processmate(`x`)` and the topmost isolated node in `(P(processmate(`x`))`. (This would have us traversing the D edges in the 'wrong' direction, i.e. creating a *reverse cycle*. By reversing the

unroll(z) ::= { y $\leftarrow$ last(P(z));
let z be the j'th node in the current loopbody of P(z);
add { $b_1, \ldots, b_j$ } to preamble(P(z));
for $i = 1$ to j do { $x_i \leftarrow$ name(newnode(b_i)) };
P(z) $\leftarrow$ P(z) $\cup$ { $x_1, \ldots, x_j$ }; N $\leftarrow$ N $\cup$ { $x_1, \ldots, x_j$ };
add-D-edges($\langle y, x_1, \ldots, x_j \rangle$) }

add-D-edges($\langle s_0, s_1, \ldots, s_k \rangle$) ::= { D $\leftarrow$ D $\cup \{\langle s_0, s_1\rangle, \langle s_1, s_2\rangle, \ldots, \langle s_{k-1}, s_k\rangle\}$ }

duplicate(P) ::= { y $\leftarrow$ last(P);
$j \leftarrow$ input-size(loopbody(P));
let $b_1, \ldots, b_j$ be the first j nodes of loopbody(P);
for $i = 1$ to j do { $x_i \leftarrow$ name(newnode(b_i)) };
loopbody(P) $\leftarrow$ loopbody(P) $\cup$ { $x_1, \ldots, x_j$ };
N $\leftarrow$ N $\cup$ { $x_1, \ldots, x_j$ };
add-D-edges($\langle y, x_1, \ldots, x_j \rangle$) }

tin(y) ::= { $z \leftarrow$ the topmost isolated node in P(y); return(z) }

first-node(α,P) ::= { $z \leftarrow$ the topmost isolated node of type α in P;
return(z) }

add-U-edge(x, v) ::= { U $\leftarrow$ U $\cup$ $\{\langle x, v \rangle\}$ }

cycle-exit(P(u)) ::= { print ("cycle found:", G, Path $\frown\langle$ P(u) $\rangle$); halt }

isolated-node-exit(x) ::= { print ("isolated node found:", G, x); halt }

Figure 3.3: The Service Subroutines

main::= { preamble $\leftarrow$ true;
while (there is an isolated preamble node u)
do { Path $\leftarrow \langle\rangle$; ptin($u$)};
preamble $\leftarrow$ false;
while (there is an isolated loop node u)
do { Path $\leftarrow \langle\rangle$; ptin($u$)}
}

Figure 3.4: The `main` Program

direction on the edges between processes, we can construct a legal cycle)[1].

3.3 Examples

In this section, we give an example illustrating the message flow graph algorithm in operation, and a further example showing the NMC for complete communication graphs (Theorem 5.2.3). The NMC is used in the proof of termination of the algorithm, but is also of interest in its own right as a necessary combinatorial condition for avoiding deadlock.

3.3.1 An Illustration of the Algorithm

Figures 3.6, 3.7, 3.8, 3.9 and 3.10 illustrate the execution of the message flow graph algorithm on a simple example. We assume that the D edges for the input graph have been added before `main` is called. Since there are preamble nodes in the input graph, `main` sets `preamble` = *true*. To illustrate the algorithm in progress, we consider the processing after the first two matched pairs, of type `W(CB)` and `W(CA)`, have already been found and added as U edges (Figure 3.6).

Since not all preamble nodes have been matched, the `while` loop arbitrarily selects an isolated preamble node, say `W(AB)` in `preamble(A)`, and calls `ptin` on this node. `A` is added to Path, which had been empty, `W(AB)` is

[1]Cycle detection can be implemented more efficiently, for example by an array of flags labelled with the process names. When a process is visited during the recursive calls, its flag is set to true. If its flag is already true, then we have deadlock and the cycle can be read directly off the parameters in the call stack. For the sake of clarity, we omit such details in the algorithm code.

```
ptin(u)::= { if P(u) ∈ Path then cycle-exit(P(u));
             Path ← Path ⌢⟨P(u)⟩;
             x ← tin(u);
             v ← match(x);
             if (preamble = true and v ∉ preamble(P(v)))
               then unroll(v);
             handle(x, v);
             Path ← Path / ⟨P(u)⟩ }

match(x) ::= { α ← event-type(x);
               if (there is no topmost isolated node of type α
                                               in processmate(x))
                 then if (preamble = true) then isolated-node-exit(x)
                        else if (there is a node of type α
                                      in loopbody(processmate(x)))
                               then duplicate(processmate(x))
                               else isolated-node-exit(x);
               z ← first-node(α,processmate(x)); return(z) }

handle(x, v) ::= { repeat ptin(v)
                     until (v = tin(v));
                   add-U-edge(x, v) }
```

Figure 3.5: The Core of the Algorithm

already the topmost isolated node in `A`, and `match` is called to find the first unmatched node of type `W(AB)` in `B`. The processmate in this case is the first node in `loopbody(B)`. Since `W(AB)` is not in `preamble(B)`, and `preamble = true`, `unroll` adds `W(AB)` to `preamble(B)`, and rotates `loopbody(B)`, putting `W(CB)` first, and then `W(AB)` second in `loopbody(B)`. `handle` is then called on the `W(AB)` nodes, with the one in `B` corresponding to `v` and the one in `A` corresponding to `x`. `handle` in turn calls `ptin` on the `W(AB)` in `B`. `B` is added to `Path`, and the second `W(CB)` in `preamble(B)`, which is now the topmost isolated node in `B`, is the argument for the next call to `match`. `match` returns the first `W(CB)` in `loopbody(C)`. Since `W(CB)` is not in `preamble(C)`, and `preamble = true`, `unroll` makes the first loop body occurrence of `W(CB)` the last preamble node of `C`, and adds a second occurrence of `W(CB)` to the end of `loopbody(C)`.

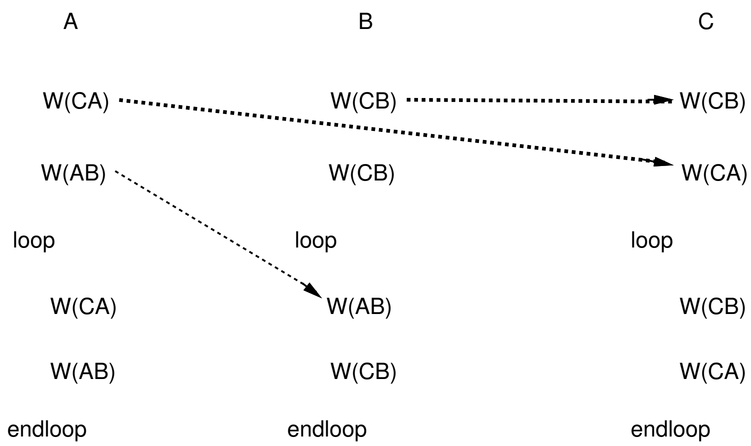
Both instances of `W(CB)` are now topmost isolated nodes, so `handle` creates the `W(CB)` U edge and returns; upon the return `ptin` deletes `B` from `Path`. `ptin` now returns, and the first call to handle continues its `repeat` loop. Since both `W(AB)` nodes are now topmost isolated nodes, the `W(AB)` U edge is created, and `handle` returns to `ptin`, which deletes `A` from `Path`, and returns to `main`. Since all the preamble nodes are now matched, `preamble` is set to *false*. The behavior of the algorithm on the loop body nodes is similar and straightforward. The result is shown in Figure 3.10.

We can consider another example by deleting the first node from each process in Figure 3.6. A cycle deadlock condition is present in this example, as illustrated in Figure 3.11. Note that the unrotated loop code in Figure 3.6 contains a cycle, but the loop rotations during the execution of the message flow graph algorithm caused the cycle to disappear. Conversely, rotations may introduce cycles. So we cannot detect cycles involving loop bodies in advance of processing the preamble nodes.

3.3.2 Loop Body Code and the NMC

An Example. We present a simple example showing how to duplicate communicating loops to make them nummatch. It also illustrates the *Numerical Matching Condition*, which is both necessary and sufficient for loop-bodies to be nummatchable. We then modify the example to show some non-nummatchable loops. We show in Theorem 5.2.3 that if infinite loop-bodies are not nummatchable, then there will be deadlock. Consequently, the NMC can be used as a quick check for deadlock.

There are only 3 processes in the example. Initially, the communication



The D edges are omitted for clarity. Only the U edges are shown.

Figure 3.6: Construction of message flow graph up to the call to ptin(x)

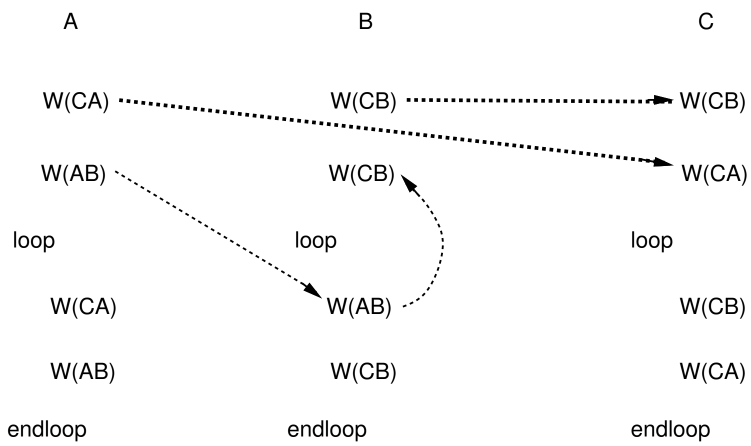


Figure 3.7: The call to handle(x,y)

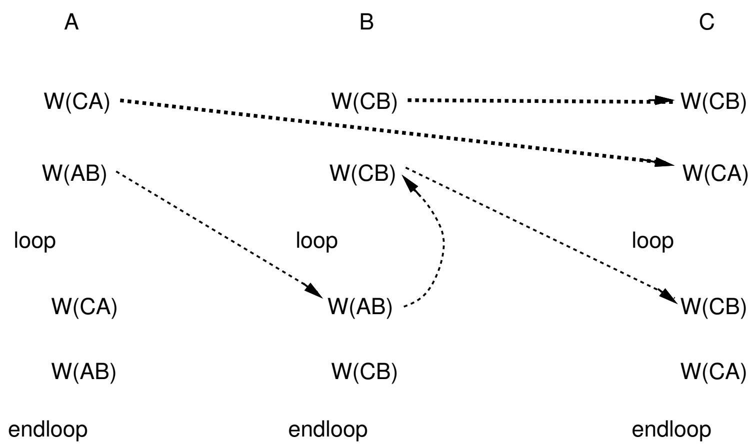


Figure 3.8: The call to ptin(z)

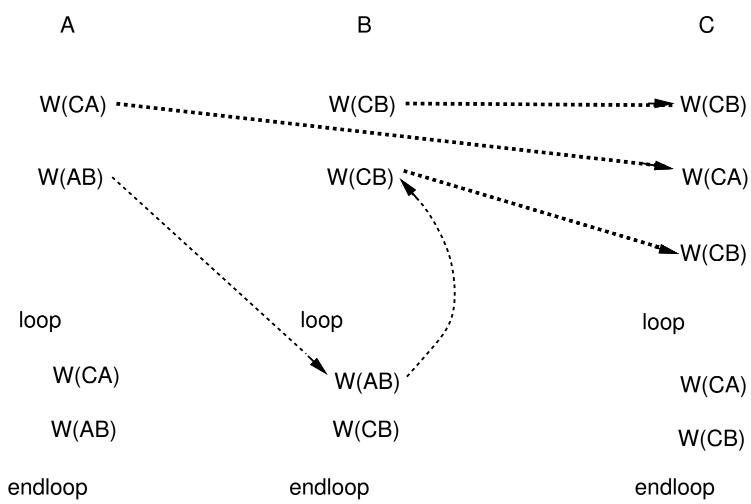


Figure 3.9: Unrolling C in handle(z,w) and returning

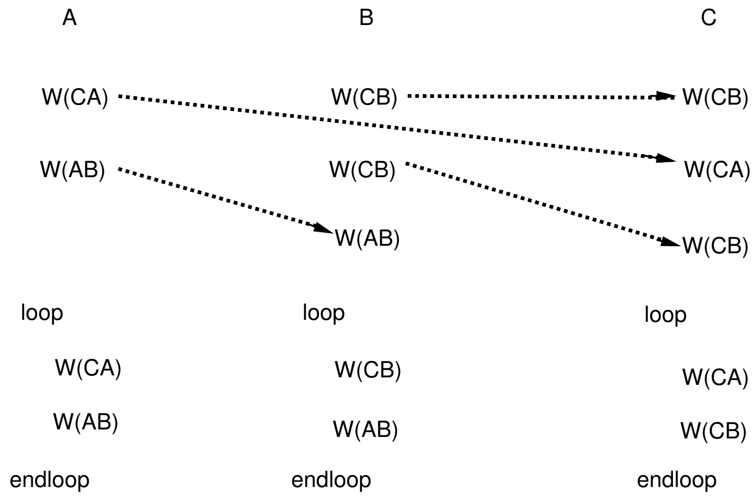


Figure 3.10: Message flow graph after return of ptin(x)

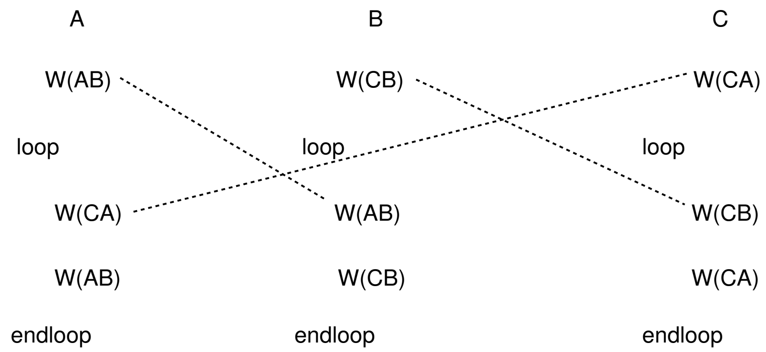


Figure 3.11: A deadlock condition in a message flow graph

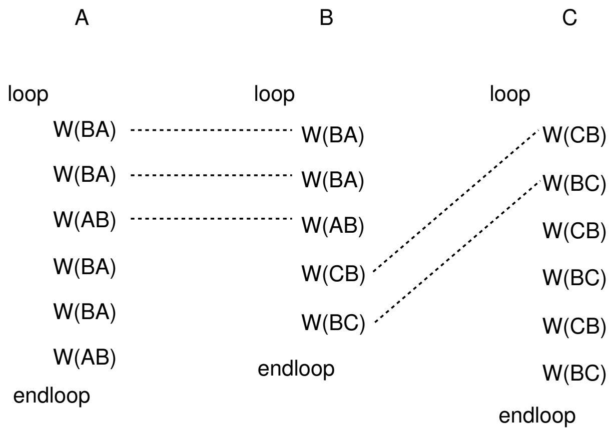


Figure 3.12: Processing loops prior to duplication of loopbodies

graph is a tree, so the loopbodies are guaranteed to be nummatchable (see lemma 5.2.5). We then consider modifications in which there are communications between all three processes: one modification in which the loopbodies are not nummatchable, and thus deadlock, and one in which they are nummatchable.

The Initial Example. Suppose that processes `A`, `B`, and `C` contain the commstats given in Figure 3.12 (we omit all purely computational code that may occur between commstats). The mfg joins with undirected edges those commstats denoting the same commact, and with directed edges those statements in the same process which are consecutive. A partial construction of an mfg for the example is shown in Figure 3.12. Since the processes do not immediately nummatch, some loopbodies must be duplicated.

The loopbody of `A` is the sequence `W(BA) W(BA) W(AB)` repeated twice. However, we cannot rewrite it as `loop W(BA) W(BA) W(AB) endloop`, because our assumption of minimum loopbody size implies that there is omitted computational code intervening between the commstats that is not identical in each occurrence of `W(BA) W(BA) W(AB)`.

After the 2-duplication of the loopbody of `B`, the loopbodies of `A` and `B` nummatch, as shown in Figure 3.13. The communications with `C` in the new loopbody of `B` are `W(CB) W(BC) W(CB) W(BC)`. If we use the 3-duplication of the new loopbody of `B`, and the 2-duplication of the original loopbody of `C`, `B` and `C` will nummatch. However, to make the loopbodies of `A` and

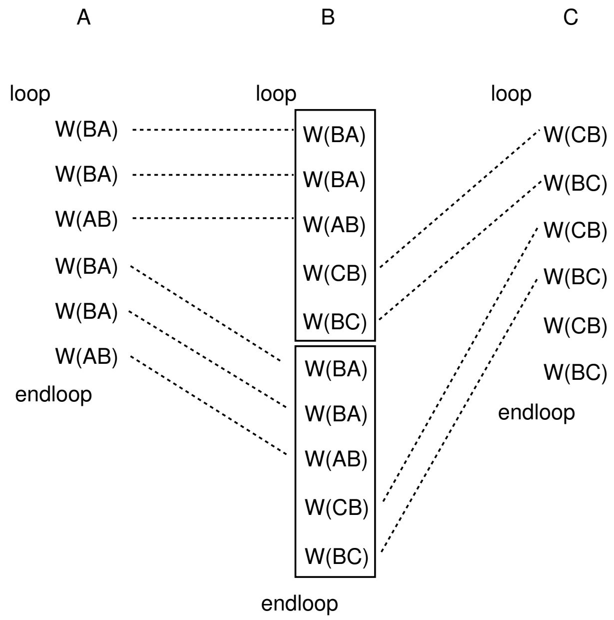


Figure 3.13: After the 2-duplication of `B`

`B` nummatch, we need to duplicate the loopbody of `A` the same number of times as `B` (i.e. 3 times). The result is shown in Figure 3.14, and is shown as a minimum system loop process in Figure 3.15 (for more information concerning minimum system loop processes, see Chapter 6). The system loop process in Figure 3.15 retains statements of the form `W(AB)` in order that the reader can correlate these statements with commstats in Figures 3.12, 3.13 and 3.14. In fact, each statement `W(XY)` must be replaced in the minimum system loop process by an equivalent computational statement. For example, if a commstat `W(AB)` represents a CSP-like remote assignment statement in which `A` writes the value of its local variable x, and `B` reads the value into its local variable y, then the computation statement actually replacing this occurrence of `W(AB)` would be for example $B.y \leftarrow A.x$, where the variables $B.x$, $A.y$ are (now global) program variables, given names reflecting the process of origin to avoid name conflicts. The assumption of minimality of loopbodies implies that there must be sequences of computation statements within the two identical commstat sequences that differ. Therefore, the loop bodies cannot be reduced in size, even though the commstats alone create a repeating pattern.

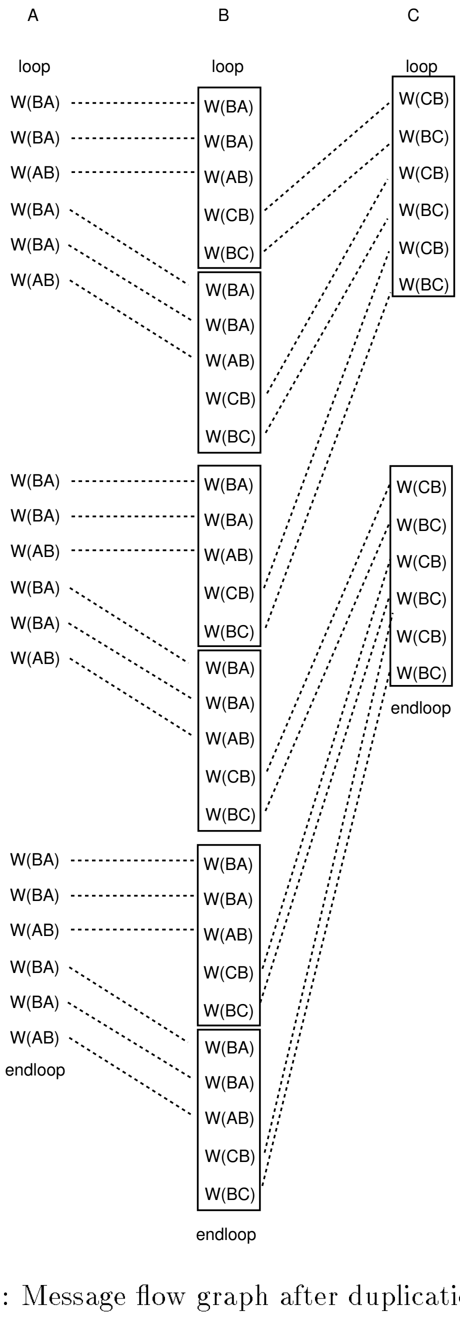


Figure 3.14: Message flow graph after duplication of loops

```
loop
W(BA);   W(BA);   W(AB);   W(CB);   W(BC);
W(BA);   W(BA);   W(AB);   W(CB);   W(BC);
W(BA);   W(BA);   W(AB);   W(CB);   W(BC);
W(BA);   W(BA);   W(AB);   W(CB);   W(BC);
W(BA);   W(BA);   W(AB);   W(CB);   W(BC);
W(BA);   W(BA);   W(AB);   W(CB);   W(BC)
endloop
```

Figure 3.15: A Minimum System Loop Process

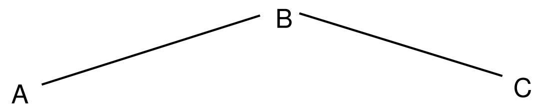


Figure 3.16: The communication graph for the example

Figure 3.16 illustrates the communication graph of Figure 3.12, which is a tree because there is no communication between `A` and `C`. In lemma 5.2.5 we prove that when the communication graph is a tree, the loopbodies are always nummatchable. Thus in this case the *Sequence Condition* (Chapter 5) is both necessary and sufficient for the existence of a matching of all the nodes in the loopbodies.

A Non-Matching Modification. Now consider the example in Figure 3.17, which in addition to the initial code of Figure 3.12 contains commstats between `A` and `C`, making the communication graph no longer a tree. In the duplicated version of the matching example (Figure 3.14), both `A` and `C` are duplicated twice. Therefore, in our modified version, the new loopbody of A contains two `W(CA)` statements, and the new loopbody of `C` contains four `W(CA)` statements. The new loopbody of `A` must be duplicated once to match with the `W(CA)` statements in the loopbody of `C`. But now the loopbodies of `A` and `B` no longer nummatch. In fact, there is no way to make all the loopbodies simultaneously nummatch, and the parallel version will eventually deadlock (see Theorem 5.2.8). This example shows that if there is a loop in the communication graph, the loopbodies are not necessarily nummatchable.

loop	loop	loop
W(BA)	W(BA)	W(CB)
W(CA)	W(CB)	W(CA)
		W(CA)
endloop	endloop	endloop
ab = 1	**bc = 1**	**ca = 2**
ac = 1	**ba = 1**	**cb = 1**

Figure 3.17: A Non-Matching Example

The example in Figure 3.17 violates the NMC (see Theorem 5.2.3). Suppose ab is the number of commstats in the original code of `A` that mention process `B` (either `W(A,B)` or `W(B,A)`), ba is the number of commstats in `B` which mention `A`, and similarly for commstats between other pairs of processes. The NMC for these three processes states that $ab.bc.ca = cb.ba.ac$. For this example, using the numbers from the original unduplicated loopbodies this yields: $6.2.ca = 6.3.ac$ or $2ca = 3ac$. Since originally $ca = ac = 0$, the equation was satisfied, and the loopbodies were nummatchable. If the original processes had contained `W(AC)` in `A` and `W(AC) W(AC)` in `C`, then $ac = 1$ and $ca = 2$, which does not satisfy the equation, and so the loopbodies are not nummatchable. If instead we had added `W(CA) W(CA)` to `A`, and `W(CA) W(CA) W(CA)` to `C`, then the NMC is satisfied and the loops are nummatchable.

Even if the Sequence Condition and the NMC are satisfied, it is still necessary to check for cycles. Figure 5.1 shows processes for which these two conditions are satisfied, and Figure 5.2 shows a cycle in these processes. Each has precisely one commstat with each of two other processes (so satisfying the NMC), which is of the right sort (satisfying the Sequence Condition), but there is a cycle in the mfg, and the processes deadlock. In this example, the cycle occurs in the original loopbodies. Figure 5.1 shows an example of processes that satisfy both conditions but in which duplication of at least one loopbody is required in order to obtain a cycle, shown in Figure 5.2.

Chapter 4

Analysis of the Algorithm

In this chapter, we prove termination and correctness of the algorithm, and also uniformity of the sequentialization problem for loop process code with convenient primitives. Statements of the main theorems may be found in Section 4.1. The reader interested primarily in the algorithms and techniques may prefer to skip the proofs, which may be found in Section 4.2. We show correctness by proving that if the code contains some *circular wait* condition (a cycle), a cycle will be detected. In particular, the combination of the Cycle Condition and the Isolated Node Condition is sufficient for detecting all deadlock. The circular wait condition, in combination with other conditions which are satisfied by our choice of communication primitives, is the standard condition for deadlock [Hol72], [SPP91] (p197).

4.1 Main Correctness and Termination Theorems

A termination by a call to `cycle-exit` or to `isolated-node-exit` is a *failure termination*, and a termination by return of all subroutine calls is a *normal termination.*

The service routines in Figure 3.3 are all straight-line code (e.g. `unroll`), calls to routines which are straight-line code (e.g. `duplicate`), or table look-ups (e.g. `first-node`). We assume without proof that these routines all terminate and behave correctly. Our main concern is the behavior of the mutually recursive routines `ptin` and `handle`, the routine `match`, which constitute the core of the algorithm, and of course the `main` program itself.

A sequence of technical lemmas lead to proofs of the following two lemmas. The proof of termination on loop bodies also makes essential use of the Numerical Matching Theorem (Theorem 5.2.3 for complete graphs, and Lemma 5.2.6 for non-complete graphs).

Lemma 4.1.1 (Termination on Preamble Nodes) *If there is no failure-termination, then eventually* `preamble = false`*, and all preamble nodes are matched to other preamble nodes.*

Lemma 4.1.2 (Termination on Loop Body Nodes) *If there is no failure-termination, then eventually all loop body nodes are matched to other loop body nodes.*

The Termination Theorem follows directly from these two lemmas.

Theorem 4.1.3 (Termination) *If the algorithm does not halt with a call to* `cycle-exit` *or* `isolated-node-exit`*, then the algorithm normal-terminates.*

The proof of correctness that correlates failure–termination with cycles in the graph and circular-wait conditions in the code, follows from a sequence of key lemmas. We show that the Cycle Condition together with the Isolated Node Condition are sufficient for detecting deadlock. We also show that loop processes with convenient primitives are uniform.

Our main theorem is that any execution trace respects the U edges, i.e. in any trace, commacts can be executed only by two statements which are connected by a U edge in the message flow graph, and not between statements that have no U edge between them (and thus are connected by U edges to other statements). We call this property *coherence*. We then show that any trace trying to execute a cycle will deadlock. We use coherence to prove that if the algorithm terminates normally, there is a non-deadlocking trace; and if failure-termination occurs that all traces deadlock. Finally, we show uniformity. Where $e = \{x, v\}$ is an edge, we shall use the phrase *e is drawn* to mean that `add-U-edge(`x, v`)` is executed.

The following theorem shows that the graph constructed satisfies the important condition in the definition of message flow graph (Section 3.1.3).

Theorem 4.1.4 (Communication respects U edges) *Suppose there is a U edge in the message flow graph from x to y. Then in all execution traces which execute x or y, the commact denoted by x is the same action as that denoted by y.*

Theorem 4.1.4 is used in showing that normal termination implies there are no deadlocking traces, which in turn is part of showing uniformity. The proofs show that cycles in the message flow graph correlate with circular wait conditions in process executions (note that the algorithm doesn't draw cycles). From this we may conclude how the algorithm behaves under failure-termination or normal-termination conditions.

Theorem 4.1.5 (No Cycles Drawn) *There are no cycles in the message flow graph as drawn by the algorithm.*

Finally, the Correctness Theorem follows from these results and other technical lemmas.

Theorem 4.1.6 (Correctness and Uniformity) *When the algorithm terminates, either it failure-terminates, in which case all traces deadlock, or it terminates normally, in which case all traces are deadlock-free, and every node is connected by precisely one U edge to another node.*

4.2 Proofs and Subsidiary Lemmas

4.2.1 Termination of the Algorithm

In this section, we state and prove the technical lemmas needed to show that the message flow-graph algorithm always terminates. The termination proof has two parts: termination on the preamble nodes (Lemma 4.1.1) and termination on the loop bodies (Lemma 4.1.2).

Lemma 4.2.1 *(a) If* `ptin` *does not failure-terminate, then the only way that a call to* `ptin` *may not return is if the call to* `handle` *does not return.*
(b) If `handle` *does not failure-terminate, then the only way that a call to* `handle` *may not return is if some call to* `ptin` *does not return.*

Proof: (a) Immediate from the code in `match`.
(b): Assume, for a contradiction, that all calls to `ptin` return in some non-returning call of `handle`(x, v). Since these calls to `ptin` all return, by (a) all calls to `handle` within any of these calls to `ptin` return. Each such call to `handle` has as its first argument some node v' above v in `P`(v); upon return of the call to `handle`, a U edge has been drawn between v' and some matching node in `processmate`(v'). Thus, on return of `handle`, v'

is no longer isolated. On return of the `ptin` call which invoked this call of `handle`, some node v' above v which was isolated on entry is no longer isolated. If all calls to `ptin` within this call of `handle`(x, v) terminate, and `handle`(x, v) does not terminate, the repeat loop in `handle`(x, v) must repeat indefinitely. Therefore, there must be infinitely many returning calls to `ptin`, which implies by the above reasoning that there must be infinitely many nodes v' above v. But there are only finitely many nodes above v. Contradiction. ∎

We now prove some invariants of the mutually recursive routines `handle` and `ptin`.

Lemma 4.2.2 *(a) If a call to* `ptin` *returns, and* $|\texttt{Path}| = k$ *on entry, then* $|\texttt{Path}| = k$ *on exit.*
(b) If a call to `ptin` *does not return, and* $|\texttt{Path}| = k$ *on entry, then in all subsequent states* $|\texttt{Path}| \geq k$*.*

Proof: (a) First, we observe that `ptin` is the only routine that changes the value of `Path`. `Path` is incremented on entry to, and decremented on exit from, `ptin`. Thus, the net length of `Path` remains the same.
(b) By Lemma 4.2.1, the only way that `ptin` cannot return is if `handle` does not return. Thus, in this case `Path` is incremented on entry, and not decremented. Any subsequent completed call to `ptin` leaves the size of `Path` invariant, by part (a). Thus, all subsequent calls to `ptin` leave the size of `Path` either the same or greater, as stated by the lemma. ∎

Lemma 4.2.3 *(a) If* `handle`(x, v) *does not return, and* `Path` = `s` *on entry, then in subsequent states there is always some* `t`*, possibly the empty string, such that* `Path` = `s`⌢`t`*.*
(b) If `ptin`(u) *does not return, and* `Path` = `s` *on entry, then in subsequent states there is always some* `t`*, possibly the empty string, such that* `Path` = `s`⌢`P`(u)⌢`t`*.*

Proof: Part (a) follows directly from Lemmas 4.2.1 and 4.2.2. Part (b) follows from (a), Lemma 4.2.1, and inspection of the code for `ptin`. ∎

Lemma 4.2.4 *If* `ptin` *is called on the same process twice, without an intervening return of* `handle`*, then* `cycle-exit` *is called.*

Proof: Suppose `ptin`(u) is called, followed by `ptin`(v), where `P`(u) = `P`(v), without an intervening return of `handle`. By Lemma 4.2.3, `P`(u) is on the path at the call of `ptin`(v). Thus, the predicate in the first statement in `ptin`(v) is true, and `cycle-exit` is called. ∎

Lemma 4.2.5 *If* `handle`(x, v) *does not return, and* `isolated-node-exit` *is not called, then there is a process* `Q`$\neq$`P(v)`*, and* $v' \in$ `Q`*, such that* `handle`(x', v') *is called within the call to* `handle`(x, v)*, and* `handle`(x', v') *does not return.*

Proof: Let `handle(-,`v`)` be the first call to `handle` that does not return. Consequently, the `repeat` loop in `handle(-,`v`)` does not return, which means that either (a) there is a call to `ptin`(u) within the `repeat` loop that does not return, where `P`(u) = `P`(v), or (b) v is never equal to `tin`(v). In case (a), by Lemma 4.2.1, there is a call `handle`(x', v') that does not return, where `P`(x') = `P`(u), and so `P`(u) = `P`(x') $\neq$ `P`(v'), and we are done. In case (b), by Lemma 4.2.4, there is an intervening return of `handle` between each return of `ptin`(u) for $u \in$ `P`(v), u above v. Each intervening return of `handle`(x', v') for $x' \in$ `P`(v) removes a topmost isolated node from P(v) by constructing a U edge. There are only finitely many isolated nodes above v, and while `ptin` is being called on nodes in P(v) above v, neither `unroll` nor `duplicate` is called on any node in `P`(v), so no new nodes are added to `P`(v). Hence, if the `repeat` loop does not terminate, it must be because some call to `ptin`(x') does not return, for some $x' \in$ P(v) in the `repeat` loop. This can happen only because the call to `handle`(x', v'), for $v' \in$ `Q`, in the non-returning call to `ptin`(x') does not return, as claimed. ∎

We now show that if neither `cycle-exit` nor `isolated-node-exit` is called, then each call to `ptin(u)` from `main` completes.

Lemma 4.2.6 *If neither* `cycle-exit` *nor* `isolated-node-exit` *is called, then every call to* `handle`*, and every call to* `ptin`*, returns.*

Proof: The `handle` and `ptin` subroutines are mutually recursive, and by Lemma 4.2.1 a call to one does not complete only if there is a call to the other that does not complete. Any other statement in either routine is straightline code, or a routine which we assume terminates, and therefore all other statements in the routines terminate. So, without loss of generality, it suffices to show that any call to `handle` completes under the hypotheses of the lemma.

We show the contraposition of the statement of the lemma. Suppose some call of `handle` does not return, and `isolated-node-exit` is never called. We shall show that `cycle-exit` is called. By lemma 4.2.5, every non-returning call engenders another non-returning call to `handle` on a different process. Since there are only finitely many processes, at some point a non-returning call to `handle`$(-, v')$ will be made on a process for which there is still an outstanding call to `handle`$(-, v)$. By lemma 4.2.3, `P`(v) is still in `Path` from the earlier call, and the call to `ptin`(v') in `handle`$(-, v')$ will detect this. Thus, `cycle-exit` will be called. ∎

Lemma 4.2.7 *If the algorithm does not failure-terminate on any iteration of either* `while` *loop in* `main`*, then it terminates normally on this iteration.*

Proof: This follows directly from lemma 4.2.6, since the body of the `while` loop call is just a call to `ptin`. ∎

It does not immediately follow from Lemma 4.2.7 that if the algorithm as a whole does not failure-terminate on the preamble, that it normal-terminates. While processing the preamble, calls to `unroll` can add nodes to some or all of the original preambles, and while processing loopbodies, calls to `duplicate` can add nodes to the original loopbodies. We need to show that these additions cannot happen unboundedly often.

Lemma 4.2.8 *Upon return of* `handle`$(-, v)$, *v is above the topmost isolated node in* `P`(v).

Proof: The `repeat` loop in `handle` exits with $v =$ `tin`(v). The immediately subsequent call to `add-U-edge` ensures that v is no longer isolated, hence v is now above the topmost isolated node in `P`(v). ∎

Some of the following lemmas specialize just to the preamble. For the lemmas which make sense also for loop bodies, similar lemmas are provable for the loop bodies, in particular Lemma 4.2.10. We do not include statements or proofs of these lemmas.

Lemma 4.2.9 *If preamble = true, then* `duplicate` *is not called.*

Proof: Immediate from the code in `match`. ∎

Lemma 4.2.10 *Let u be an isolated preamble node on which an iteration of the* `while` *loop is called. If the* `while` *loop iteration completes on u, then the total number of preamble isolated nodes has been reduced.*

Proof of 4.2.10: Suppose not. We shall derive a claim from this assumption, in order to obtain a contradiction.

Claim: There is a new isolated node in some preamble.

Proof of Claim: If the total number of isolated nodes is not reduced, then it either (a) increases, or (b) stays the same. The claim is obviously true in case (a). In case (b), the return of `ptin(`u`)` requires that `handle(`x`,`v`)` has returned, where x is the topmost isolated node in `P(`u`)` at the beginning of the call, and $v =$ `match(`x`)`. The return of `handle` implies directly that `add-U-edge(`x, v`)` has completed, which means that x is no longer an isolated node in `preamble(P(`x`))` upon the return of `ptin(`u`)`. Thus, since the total number of preamble isolated nodes stays the same, there must be some other node y that was in `loopbody(P(`y`))` at the call `ptin(`u`)`, and upon return of `ptin(`u`)` is now in `preamble(P(`y`))`, and is isolated. ▮Claim.

Let y be such a new isolated node in some preamble. By Lemma 4.2.9, the node y can have moved to `preamble(P(`y`))` only by an execution of `unroll(`v`)` for some node v in `P(`y`)` such that y is above v or $y = v$. The succeeding statement to this `unroll(`v`)` is a call to `handle(`x', v`)` for some x' such that $v =$`match(`x'`)`, which by hypothesis returns. By Lemma 4.2.8, v is above the topmost isolated node in `P(`v`)`, and y is above v or $y = v$, and so y is above the topmost isolated node in `P(`v`)`. However, y is an isolated node. Contradiction. ▮

Lemma 4.2.11 *If the algorithm does not failure-terminate on the preamble, then it normal-terminates on the preamble.*

Proof: By Lemma 4.2.7, each iteration of the `while` loop terminates normally. Thus, the only possibility for non-termination is that there could be infinitely many iterations of the `while` loop, each of which terminates normally. But by Lemma 4.2.10, each normal termination of an iteration reduces the total number of preamble isolated nodes. So if there are k such nodes in the input processes, then it follows from Lemma 4.2.10 that there can be at most k iterations of the `while` loop in `main`. ∎

Lemma 4.2.12 *If the preamble* `while` *loop of* `main` *terminates normally, then all preamble nodes are matched.*

Proof: The preamble `while` loop can terminate normally only if the loop predicate eventually becomes false, i.e. there are no more isolated nodes in the preamble. Nodes that are not isolated are matched, by definition. ∎

Lemma 4.2.13 *If* `preamble = true` *and* `handle(`x, v`)` *is called, then* $x \in$ `preamble(P(`x`))`.

Proof: Suppose not. Let `preamble = true`, and let `handle(`x, v`)` be the first call to `handle` in which $x \notin$ `preamble(P(`x`))`. This call comes from a call to `ptin(`u`)`, where u is either *(a)* a topmost isolated node chosen by the `while` loop in `main`, or is *(b)* `tin(`v'`)` at some point in a previous call `handle(`x', v'`)`. In case *(a)*, x is above u, which is a preamble node, and hence also $x \in$ `preamble(P(`x`))`. Contradiction. In case *(b)*, $x' \in$ `preamble(P(`x'`))` by hypothesis, and prior to the call to `handle(`x', v'`)`, `unroll(`v'`)` is executed, so that (by direct inspection of the code for `unroll`), $v' \in$ `preamble(P(`v'`))`. We have $x =$`tin(`u`)`, so $x = u$ or x is above u. Since u is above v', this implies that x is above v', and hence $x \in$ `preamble(P(`x`))`. Contradiction. ∎

Lemma 4.2.14 *U edges are drawn from preamble nodes only to other preamble nodes.*

Proof: Nodes are matched in `ptin`, and U edges are drawn by a call to `handle(`x, v`)`. v `= match(`x`)` is unrolled immediately before this call if `preamble = true`, hence $v \in$ `preamble(P(`v`))`. If `preamble = true`, then also $x \in$ `preamble(P(`x`))` by Lemma 4.2.13. Hence both x and v are in their respective preambles. ∎

Corollary 4.2.15 *U edges are drawn from loop body nodes only to other loop body nodes.*

Proof: Immediate from Lemma 4.2.14. ∎

Proof of Termination on Preamble Nodes: The proof of Lemma 4.1.1 follows directly from Lemmas 4.2.11, 4.2.12, and 4.2.14, with the additional observation that if the preamble *while* loop terminates normally, then the next statement executed is `preamble = false`. ∎

Proof of Termination on Loop Bodies: The proof of Lemma 4.1.2 makes essential use of the Numerical Matching Condition (Theorem 5.2.3 and Lemma 5.2.6).

By Lemma 4.1.1, eventually we finish processing preamble nodes. Since `preamble` = *false* during the processing of isolated loop body nodes, `unroll` is not called by `ptin`. `Duplicate` may be called by `match` if there is no isolated node of the right type to match with. There must be a node of the right type in the loop body, else `isolated-node-exit` is called, contrary to hypothesis. `Duplicate` adds precisely one copy of the *original* (rotated) loop body to the end of the current loop body, so it increments the number of duplications of the original loop body by one. Theorem 5.2.8 implies that the Numerical Matching Condition is met, else there is a cycle. The Numerical Matching Condition implies that after a certain number of duplications of each loop body (perhaps different for each loop body), for any communication type, the numbers of nodes of that type in both processes are equal. `duplicate` is called only when there is no node to match with, and one is needed, and `duplicate` adds only one copy of the original body to the current loop. It follows from the Numerical Matching Condition that dynamic duplication will terminate after the same number of duplications is reached. With this in mind, we shall modify the algorithm so that we can more easily establish termination.

We modify our approach to use static duplication before applying the loop body processing. Prior to processing the loop bodies, we perform the number of loop body duplications required, based on the Numerical Matching Condition. We claim without proof that the semantics of the loop bodies remains the same, i.e. the new loop bodies have exactly the same collection of traces as the old. Further, during execution of the message flow graph algorithm, while there is an isolated node in any (new) loop body, there is an isolated node of the right type still in the (new) `processmate`, since both processes have equal numbers of nodes of that type. Hence the `else if` clause of `match` is never taken, and there is no dynamic duplication. Since there is no addition of nodes, the total number of nodes remains constant. Each return of `handle` reduces the number of isolated nodes by adding a U edge before each return, and no new nodes are created. Thus, the `while` loop terminates when there are no more isolated nodes to process. At termination, all loop nodes are connected by U edges to other nodes (by the termination condition), and by Corollary 4.2.15, loop nodes are connected only to other loop nodes. ▮

Proof of the Termination Theorem: The proof of the Termination Theorem 4.1.3 is a summary of previous results. By Lemma 4.2.11, if there is no failure-termination, then the algorithm normal-terminates on the preamble. The second `while` loop of `main` terminates, and all loop body nodes are matched, by Lemma 4.1.2. ▮

4.2.2 Correctness and Uniformity

We now turn to the proof that the algorithm does what we want it to do, namely it discovers cycles corresponding to deadlocks, and draws edges corresponding to pairs of commstats that always communicate only with each other, and that every trace is an extension of the natural partial ordering induced on these U edges by the D edges of the processes.

Lemma 4.2.16 *For any U edge $e = \{x, y\}$ in the message flow graph, at the time that e is drawn by* `handle`*, x and y are topmost isolated nodes in* `P(`x`)` *and* `P(`y`)` *respectively.*

Proof: Follows directly from inspection of the algorithm code. ▮

Lemma 4.2.17 *If $x, y \in$* `P`*, and x is above y, then in all traces (a) if x is a preamble node, then the action denoted by x occurs before the action denoted by y; (b) if x is a loop body node, then the action denoted by x in the m'th iteration of the loop body occurs before the action in the m'th iteration denoted by y.*

Proof: Immediate, from the meaning of the serial code defining `P`. ▮

Proof that Communication Respects U edges: We prove Theorem 4.1.4 by contradiction. Suppose not. Let T be an execution trace for which there is an U edge $e = \{x, y\}$ in the message flow graph, but x and y denote different actions in T. Further, let e be the first such edge drawn by `handle` during execution of the message flow graph algorithm, and suppose WLOG that x is executed by T before y. If x is in the preamble, let a be the action in T that is denoted by x but not by y. If x is in the loop body, let a be the first action in T that is denoted by x but not by y.

If x is in the loop body, we are going to simplify the argument by moving x into the preamble by unrolling the loop body. In performing the unrolling, we make use of the fact that edges are drawn from preamble nodes only to preamble nodes (Lemma 4.2.14), and from loop body nodes only to loop body nodes (Corollary 4.2.15).

Let a occur on iteration m of the loop body of `P(`x`)`. We first perform an unrolling of the loop body of `P(`x`)` m times. In order to retain the property stated in Lemma 4.2.14 and its equivalent for loop bodies, we need to unroll other loop bodies also. One way of doing it is to run the message flow graph algorithm again with all input processes the same, except for `P(`x`)`. If the original `P(`x`)` was (α; `loop` β `endloop`), where $\beta = \gamma^\frown\delta$ and $a =$ `last(`γ`)`, then the new input process is `P'(`x`)`::= $\alpha^\frown(\beta^{m-1})^\frown\gamma$; `loop` $\delta^\frown\gamma$ `endloop`. We claim without proof that `P'(`x`)` and `P(`x`)` have identical execution traces, and thus that this holds for the two input systems also. We further claim on this basis, without proof, that the algorithm normal-terminates (failure-terminates) on the new input if and only if it normal-terminates (failure-terminates) on the old input. The results in the last section hold of the new input. There are now m copies of the original (loopbody) node x in the preamble. We let x denote the last node of all these copies (i.e the mth copy of the original x), and let y denote its match. Thus we can assume that a is an action denoted by the preamble node x in the message flow graph. Let b be the action in T denoted by y (the node that x is connected to by a U edge).

By Lemma 4.2.17, all actions in `P(`x`)` and `P(`y`)` above x and y occur in T before the actions denoted by x and y. Further, these actions occur as specified by the U edges, by hypothesis.

Since by hypothesis x is executed by T, the action denoted by x must be the same as that denoted by some y' in `P(`y`)`. All nodes with the same event type as y (and y') above y in `P(`y`)` are joined by U edges, by hypothesis, to nodes above x in `P(`x`)`. Also by hypothesis, all these U edges denote actions as executed by T. By Lemma 4.2.16, when x is matched with y in the message flow graph algorithm, both are topmost isolated nodes. Hence y is above y'. So when T executes x, all nodes above y of same event type as y (and x) have been executed, and y is the first unexecuted statement of the right event type in `P(`y`)`. If y' is now executed (since it is the same action as denoted by x), then y can never be executed, contrary to the meaning of the serial program `P(`y`)`. ∎

We need to show that the conditions we identify as deadlocking conditions, namely the conditions corresponding to `isolated-node-exit` and `cycle-exit`, imply that any execution trace will deadlock. Further, if the algorithm normal-terminates, we must show that no execution trace deadlocks. These two conditions imply directly that loop processes using convenient primitives are uniform.

Lemma 4.2.18 *If the algorithm calls* `isolated-node-exit(`x`)`*, then all traces which reach x have a deadlock condition with* `P(`x`)` *waiting at x.*

Proof: Suppose the system is executed with trace T which reaches x. T respects all U edges in the message flow graph by Theorem 4.1.4. When `isolated-node-`
`exit(`x`)` is called, x is topmost isolated node in `P(`x`)`. Thus if the trace reaches x, all nodes above x have been executed. x has no match, and thus `P(`x`)` executes an indefinite wait at x. ∎

Lemma 4.2.19 *If the algorithm calls* `cycle-exit(`x`)`*, then any trace which reaches x has a deadlock condition with* `Path` *containing a cycle of processes including* `P(`x`)` *which are involved in a circular wait.*

Proof Sketch: As in the proof of Lemma 4.2.18, if the system executes, it executes consistently with the U edges drawn already. Suppose the algorithm calls `cycle-exit(`x`)`. Let T be a trace which reaches x. It is easy to reach a contradiction under the supposition that x is executed. In particular, we have indicated informally in Chapter 3 how a cycle cannot be executed, and it is trivial to turn this into a more rigorous proof. Thus, x is not executed, and since all nodes above are executed (by supposition), `P(`x`)` waits indefinitely on x. ∎

Theorem 4.1.5 shows that there are no cycles that get drawn inadvertently during execution of the message flow graph algorithm. To prove the theorem, we Lemma 4.2.20, in which we assume that paths may contain edges in the transitive closure of D, which we call D^*. Clearly this assumption is without loss of generality. A *reverse path* or *reverse cycle* is a path or cycle in the message flow graph in which the D edges are traversed in the opposite direction from which they are drawn.

Lemma 4.2.20 *Suppose that* `handle(`x_0, y_0`)`*, ..., * `handle(`x_{k-1}, y_{k-1}`)` *are consecutive recursive calls to* `handle` *i.e. no earlier call returns before a later call is made. Further, suppose all these calls return. Then* $C = x_0, y_0, ..., x_{k-1}, y_{k-1}$ *is a reverse path in the message flow graph.*

Proof: Since the calls are consecutive and recursive, each call `handle(`x_{j+1}, y_{j+1}`)` for $0 < j < k-1$ is invoked by a call `ptin(`y_j`)` in the `repeat` loop of `handle(`x_j, y_j`)`. Thus `P(`y_j`)` = `P(`x_{j+1}`)`, and $\langle x_{j+1}, y_j \rangle$ is a D^* edge. The U edges $\{x_{k-1}, y_{k-1}\}$, $\{x_{k-2}, y_{k-2}\}$, ..., $\{x_0, y_0\}$ are drawn consecutively, and if the U edges are interleaved with the D^* edges with the same endpoints, a path is formed. ∎

Proof that No Cycles are Drawn: We prove Theorem 4.1.5 by contradiction. Since we call subroutines in the order of a reverse cycle in the message flow graph algorithm, notation is simplified by considering reverse cycles. Suppose $C = x_0, y_1, x_1, y_2, x_2, ..., y_{k-1}\ x_{k-1}, y_0, x_0$ is a reverse cycle, where `P(`y_i`)` = `P(`x_i`)` and x_i is above y_i, and $\{x_i, y_{i+1 mod\ k}\}$ is a U edge, for each $i < k$. For brevity, we shall write x_{i+1} as shorthand for $x_{i+1 mod\ k}$. We may assume WLOG that each edge $\{x_i, y_{i+1}\}$ is such that x_i is the highest node in `P(`x_i`)` which is in a cycle. (We omit the straightforward argument for this claim). Since the U edges are drawn, at some point each x_i must have been topmost isolated node in `P(`x_i`)`. WLOG, suppose x_0 is the first x_i processed by a call `ptin(`z`)` with $z \in$ `P(`x_0`)`.

Claim: `handle(`x_{i+1}, y_{i+2}`)` is called by a call to `ptin(`y_{i+1}`)` in the `repeat` loop of `handle(`x_i, y_{i+1}`)` during execution of the message flow graph algorithm, for each $i < k$.

Proof: Suppose not, and let i be the least index for which `handle` is not called in the order specified. Consequently, x_{i+1} is never topmost isolated node in `P(`x_{i+1}`)` during the call to `ptin(`y_{i+1}`)` in the `repeat` loop in `handle(`x_i, y_{i+1}`)`. Thus either (a): the U edge $\{x_{i+1}, y_{i+2}\}$ was drawn, during execution of a call `ptin(`z`)` within that `repeat` loop, by some other call `ptin(`z'`)` for z' ∈`P(`x_{i+1}`)`; or (b): the U edge $\{x_{i+1}, y_{i+2}\}$ was drawn *before* the call to `handle(`x_i, y_{i+1}`)`.

In case (a), `P(`z`)` (= `P(`x_{i+1}`)`) is placed in `Path` by the call to `ptin(`z`)` and remains there until the return of `ptin(`z`)`, by Lemma 4.2.3. So `P(`z`)` ∈ `Path` during the call of `ptin(`z'`)`, whose first statement detects that `P(`z'`)` = `P(`z`)` ∈ `Path`, and calls `cycle-exit`. Thus `ptin(`z`)` does not return, hence the `repeat` loop in `handle(`x_i, y_{i+1}`)` does not return, and the U edge $\{x_i, y_{i+1}\}$ is not drawn, contrary to hypothesis. So case (a) cannot happen.

In case (b), x_0 is, by hypothesis, the first of the x_i for which `handle(`$x_i, -$`)` is called, and we know that $\{x_{i+1}, y_{i+2}\}$ is drawn *before* the call to `handle(`x_i, y_{i+1}`)`. Further, we know that since i is the first index to violate the claim, the edge $\{x_{i+1}, y_{i+2}\}$ is drawn after the call to `handle(`x_j, y_{j+1}`)` and before the call to `handle(`x_{j+1}, y_{j+2}`)` for some $j < i$. Since j satisfies the claim, by hypothesis, the call to `handle(`x_{j+1}, y_{j+2}`)` occurs as a result of a call to `ptin(`y_{j+1}`)` during the `repeat` loop in `handle(`x_j, y_{j+1}`)`, when x_{j+1} is topmost isolated node in `P(`y_{j+1}`)`. So $\{x_{i+1}, y_{i+2}\}$ is drawn during a call to `ptin(`y_{j+1}`)` during the `repeat` loop in `handle(`x_j, y_{j+1}`)`, when some z_{j+1} above x_{j+1} is topmost isolated node. The call `handle(`$z_{j+1}, -$`)` returns, so the edge $\{z_{j+1}, -\}$ is drawn, and by Lemma 4.2.20, the sequence of edges drawn by the algorithm from $\{x_{i+1}, y_{i+2}\}$ to $\{z_{j+1}, -\}$ forms a path. If the reverse path in C from $\{x_{j+1}, y_{j+2}\}$ to $\{x_{i+1}, y_{i+2}\}$ is replaced by this other reverse path from $\{z_{j+1}, -\}$ to $\{x_{i+1}, y_{i+2}\}$, we have a new reverse cycle C' in which one edge $\{z_{j+1}, -\}$ is above the edge $\{x_{j+1}, y_{j+2}\}$ in C, contrary to our choice of C as having edges above those in other reverse cycles. ▮Claim.

Following the claim from x_0 to x_k, we see that `handle(`x_0, y_1`)` is still in its repeat loop when `handle(`x_{k-1}, y_0`)` is called. Thus `P(`x_0`)` ∈ `Path`, and the first call to `ptin(`y_0`)` in the `repeat` loop of `handle(`x_{k-1}, y_0`)` will note that `P(`y_0`)` = `P(`x_0`)` ∈ `Path`, and `cycle-exit` is called. In particular since the algorithm is still in the `repeat` loop of `handle(`x_0, y_1`)`, the edge $\{x_0, y_1\}$ never gets drawn, contrary to our hypothesis that it is in the cycle C. This contradiction finishes the proof. ▮

We use this Theorem 4.1.5 to establish one half of uniformity.

Lemma 4.2.21 *If the algorithm terminates normally, there does not exist a deadlocking trace of the system.*

Proof: We shall argue by contradiction. Suppose there exists a deadlock in trace T, i.e. a circular wait on actions denoted by nodes $x_0, ..., x_{k-1}$, but that the algorithm has terminated normally on the system. As previously, we omit *mod* k in indices, so $x_k = x_0$, etc. Then when the circular wait is encountered, all edges drawn on nodes above $x_0, ..., x_{k-1}$ have been executed as actions by T by Theorem 4.1.4. All nodes x_i for $i < k$ are part of U edges, by Lemmas 4.1.1 and 4.1.2. Let $\{x_i, y_{i+1}\}$ be the U edge containing x_i, for $i < k$. Since $x_0, ..., x_{k-1}$ are a circular wait, $\mathtt{P}(y_{i+1}) = \mathtt{P}(x_{i+1})$ for $i < k$. All nodes above x_{i+1} have been executed by T, and the execution respects the U edge $\{x_i, y_{i+1}\}$ by Theorem 4.1.4. If y_{i+1} were above x_{i+1}, the action represented by the edge $\{x_i, y_{i+1}\}$ would have been executed already, and thus the node x_i would have been executed, contrary to supposition. Similarly, y_{i+1} cannot be the same node as x_{i+1}, else the edge $\{x_i, y_{i+1}\} = \{x_i, x_{i+1}\}$ could be executed, and could not thus be involved in a circular wait. So x_{i+1} is above y_{i+1}, and so $x_0, y_1, x_1, ..., y_{k-1}, x_{k-1}, y_0, x_0$ is a reverse cycle (with reverse D^* edges). By Theorem 4.1.5, there can be no such cycle. Contradiction. ∎

We now summarize these results in the proof of the Correctness Theorem, Theorem 4.1.6.

Proof of the Correctness and Uniformity Theorem: The statement of Theorem 4.1.6 is a summary of results. By Theorem 4.1.3, the algorithm always either failure-terminates or normal-terminates. If the algorithm terminates normally, then all traces are deadlock-free by Lemma 4.2.21, all preamble nodes are connected to other preamble nodes by Lemma 4.1.1 and all loop nodes to other loop nodes by Lemma 4.1.2. So all nodes are matched. If the algorithm failure-terminates, then by Lemmas 4.2.18 and 4.2.19, all traces deadlock. ∎

Chapter 5

Loop Analysis

We state the results in this chapter in terms of *blocks* of code, where we define a block to be a sequence of commstats. The results on blocks will be applied to the code contained in the infinite loops. We analyze loop code by considering two conditions:

The Sequence Condition tests whether communications between two processes will match (thus checking 2-process deadlock, i.e. a `W(AB)` can communicate with a `W(AB)` but not a `W(BA)`).

The Numerical Matching Condition tests if a set of loopbodies is nummatchable, by analysing the numbers of communications in each loopbody.

5.1 The Sequence Condition

The Sequence Condition applies to infinite loops and finite loops that are executed at least twice. Two blocks of code α and β that synchronize at their initial points will not deadlock pairwise if and only if the blocks define the same sequence of commacts. The sequences of with-β commstats in block α are interleaved in α with commstats communicating with other processes, and similarly with with-α commstats in β.

The Sequence Condition can also be used to analyze processes which use k-process acceptable primitives as well as those using convenient primitives. We state the Sequence Matching Theorem in its full generality, so that it applies to both cases, as well as to finite loops and loops that do not synchronize at their ends.

We say $\alpha < \beta$ to mean that the block α is an initial subsequence of the block β. $\langle\rangle < \alpha$, for $\langle\rangle$ the empty subsequence, and α any non-empty subsequence. We use the terminology $\beta = r.\alpha$ to indicate that the block β consists of r consecutive copies of the block α. If β consists of a copy of α followed by (concatenated with) a copy of γ, we say $\beta = \alpha + \gamma$. Suppose r copies of α are identical to s copies of β with a block γ at the end, i.e. $r.\alpha = s.\beta + \gamma$. The Sequence Matching Theorem states that α and β are iterated copies of some common subsequence σ, and that the length of σ can be taken to be the greatest common divisor of the lengths of α and β. The Sequence Condition below is the condition that corresponds to the theorem.

Theorem 5.1.1 (Sequence Matching) *Suppose α and β are blocks such that $r.\alpha = s.\beta + \gamma$, where $\gamma < \beta$, and integers $r, s > 1$. Then there is a sequence σ such that $\alpha = x.\sigma$, $\beta = y.\sigma$, and $\gamma = p.\sigma$, for some integers x, y, p. Furthermore, σ can be taken to be of length gcd(length(α), length(β)).*

The Sequence Condition: $\alpha = x.\sigma$, and $\beta = y.\sigma$, for some integers x, y, where σ is the subsequence whose length is the greatest common divisor of the lengths of α and β.

In Figures 3.12 and 3.14, we have implicitly illustrated the *Sequence Condition*. Suppose in the example we abbreviate `W(AB)` by 1, and `W(BA)` by 0. Then `A`'s communications with `B` are 110110, and `B`'s with `A` are 110. The common repeating subsequence is 110. Similarly, `B`'s communication with `C` is 10, and `C`'s with `B` is 101010. The common subsequence is 10.

The proof of the theorem may be found in Section 5.3. After we had formulated and proved the theorem, it was pointed out to us that proofs follow from results in formal language theory [Nar91], and number theory [Haf92]. To our knowledge, ours is the simplest elementary proof of this result, therefore we deemed it worth including here. We now give a simple corollary and an example to illustrate the practical application of the theorem.

Corollary 5.1.2 *If the lengths of α and β are relatively prime, then both are sequences containing occurrences of a single symbol only.*

Proof of the Corollary: The gcd of the processes is 1. If σ has length $gcd(\alpha, \beta) = 1$, then σ contains a single symbol. ∎

Example. To apply the corollary, consider two loopbodies

```
A: loop W(AB) W(AB) W(BA) W(AB)
B: loop W(AB) W(AB) W(BA)
```

Since the lengths of `A` and `B` are 4 and 3, which are relatively prime, and since they are not constant sequences, they will deadlock.

A consequence of Theorem 5.1.1 is that the problem of determining the minimum length repeating subsequence of α and β is reduced to computing the minimum length subsequence of the initial segments of α and β of length $\gcd(\text{length}(\alpha), \text{length}(\beta))$.

Lemma 5.1.3 *Checking the Sequence Condition is linear time in the total number of commstats in β_1 and β_2.*

Proof: First, calculate the gcd and detect the string σ in, say β_1. Verifying that β_1 and β_2 are repeating copies of σ requires only one pass. ∎

Theorem 5.1.4 *Checking the Sequence Condition for the loopbodies of n communicating processes is linear in the total number N of commstats.*

Loop Fusion, Sequentialization, and the Sequence Condition. Suppose we are trying to match the loopbody `loop` β_i `endloop` of $\mathtt{P}_i$, $i \leq i \leq m$. We need to match the β_is exactly with each other, so that we may interleave them and enclose the whole interleaving in a single `(loop...endloop)`, for sequentialization. This is similar to what is known as *loop fusion* in compiler optimization. Our version is equivalent to constructing a minimum system loop process. Each β_i must appear in the interleaving an integral number of times, so that when the loop iterates it will generate an infinite sequence of β_i traces. To match the loops, between β_i and β_j, we must have the relation $k_{ij}.\beta_i = k_{ji}.\beta_j$ for some positive integers k_{ji} and k_{ij}. Thus β_i and β_j satisfy the conditions of the theorem above with $\gamma = \langle\rangle$.

5.2 The Numerical Matching Condition

In this section, we state the Numerical Matching Condition. A few shorter proofs are included in the body of the text, as is the proof of the NMC for complete communication graphs, since we believe its elegance sets it apart from the more technical proofs. The more technical proofs may be found in Section 5.3.

5.2.1 The NMC for Complete Communication Graphs

The NMC, which is a set of linear equations, is both necessary and sufficient for a set of loopbodies to be nummatchable. Together, the Sequence Condition and the NMC are "fast" tools for deadlock detection that do not require construction of the message flow graph.

Let b_{ij} be the number of communication statements in β_i that communicate with β_j. We show how to duplicate n loopbodies so that the resulting loopbodies nummatch for $n = 2$, $n = 3$, and finally for arbitrary n when the communication graph is complete (the n-process NMC). The non-complete case is analyzed in Section 5.2.2.

Lemma 5.2.1 *Let β_1 and β_2 be the loopbodies of* $\mathtt{P}_1$ *and* $\mathtt{P}_2$*, respectively. Then, if the Sequence Condition holds for β_1 and β_2 (i.e. $\beta_1 = x.\sigma$ and $\beta_2 = y.\sigma$), β_1 and β_2 are nummatchable.*

Proof: Let $\alpha_1 = b_{21}.\beta_1$ and $\alpha_2 = b_{12}.\beta_2$. Then, since by definition the number of commstats involving β_2 in β_1 is b_{12} and *mutatis mutandis* for β_2, we have $a_{12} = b_{12}.b_{21} = a_{21}$. Therefore, the two α blocks nummatch. Since by assumption the Sequence Matching Condition holds, there cannot be any cycles involving only commstats of $\mathtt{P}_1$ and $\mathtt{P}_2$. ∎

We explicitly state the condition that must hold if three loopbodies are nummatchable. If the communication graph is complete, the condition for n loopbodies is a collection of three-loopbodies conditions; if the communication graph is not complete, the corresponding condition is a generalization of the condition for a complete communication graph.

Lemma 5.2.2 *Three loopbodies $\beta_1, \beta_2, \beta_3$, are nummatchable if and only if*

$$b_{12}.b_{23}.b_{31} = b_{13}.b_{32}.b_{21}$$

We call this condition $(\dagger)_{\beta_1\beta_2\beta_3}$, and we call similar conditions $(\dagger)$-conditions.

Proof: First, we show *sufficiency*. Suppose $b_{12}.b_{23}.b_{31} = b_{13}.b_{32}.b_{21}$. Let

$\alpha_1 = b_{31}.b_{23}.\beta_1$

$\alpha_2 = b_{13}.b_{32}.\beta_2$

$\alpha_3 = b_{13}.b_{23}.\beta_3$

We show that the α_i nummatch. α_1 and α_2 nummatch if and only if $a_{12} = a_{21}$, i.e.

$$b_{31}.b_{23}.b_{12} = b_{13}.b_{32}.b_{21}$$

which, rearranging the products, gives

$$b_{12}.b_{23}.b_{31} = b_{13}.b_{32}.b_{21}$$

which is exactly the condition in the lemma. The conditions for the other two nummatches are

$$b_{31}.b_{23}.b_{13} = b_{13}.b_{23}.b_{31}$$

and

$$b_{13}.b_{32}.b_{23} = b_{13}.b_{23}.b_{32}$$

which are both trivially true. Hence, the condition of the theorem is sufficient for the blocks to be nummatchable.

For *necessity*, suppose

$\alpha_1 = k.\beta_1$
$\alpha_2 = x.\beta_2$
$\alpha_3 = y.\beta_3$

nummatch. Then

$a_{12} = k.b_{12} = a_{21} = x.b_{21}$
$a_{13} = k.b_{13} = a_{31} = y.b_{31}$
$a_{23} = x.b_{23} = a_{32} = y.b_{32}$

by hypothesis. So

$x/k = b_{12}/b_{21}$
$x/y = b_{32}/b_{23}$
$y/k = b_{13}/b_{31}$

So

$b_{12}/b_{21} = x/k = (x/y).(y/k) = (b_{32}.b_{13})/(b_{23}.b_{31})$, and the condition follows by cross-multiplying. ∎

We now derive a necessary and sufficient condition for n loopbodies $\beta_1, ..., \beta_n$ to be nummatchable when the communication graph is complete.

Theorem 5.2.3 (Numerical Matching for Complete Comm Graphs) *Assume the communication graph for a collection of loopbodies is a complete graph, and let $\beta_1, ..., \beta_n$ be the loopbodies. A necessary and sufficient condition for $\beta_1, ..., \beta_n$ to be nummatchable is that the conditions $(\dagger)_{\beta_i\beta_j\beta_n}$ are satisfied, for each choice of i, j such that $1 \leq i < j < n$.*

Proof: First, *necessity* is trivial, since if the n blocks nummatch, then any three blocks of the n must nummatch *a fortiori*, and the ($\dagger$)-condition is just the condition from the previous lemma that three blocks nummatch.

For *sufficiency*, suppose $(\dagger)_{\beta_i\beta_j\beta_n}$ for $1 \leq i < j < n$ are satisfied. We choose multipliers λ_i, $i \leq n$, for the loopbodies so that the resulting blocks nummatch. Let $\lambda_i = (a_1.a_2.....a_{n-1})$, where $a_j = b_{jn}$ for $j \neq i, j \neq n$ and $a_i = b_{ni}$. Let $\gamma_i = \lambda_i.\beta_i$. We show that the blocks γ_i nummatch.
Consider the nummatching of γ_i and γ_j, for $i, j \neq n$. $\gamma_i = (a_1.a_2.....a_{n-1}).\beta_i$ and $\gamma_j = (c_1.c_2.....c_{n-1}).\beta_j$, where $a_j = b_{jn}$ for $j \neq i, j \neq n$, $a_i = b_{ni}$ and $c_k = b_{kn}$ for $k \neq j, k \neq n$, $c_j = b_{nj}$. Note that $c_k = a_k$ for $k \neq i, j$. The blocks γ_i and γ_j nummatch if and only if:

$$a_1.....a_{n-1}.b_{ij} = c_1.....c_{n-1}.b_{ji}$$

For $k \neq i, j$, we may cancel a_k on the left with c_k on the right of the equation, and we are left with

$$b_{ni}.b_{jn}.b_{ij} = b_{in}.b_{nj}.b_{ji}$$

and rearranging this gives

$$b_{ij}.b_{jn}.b_{ni} = b_{in}.b_{nj}.b_{ji}$$

which is $(\dagger)_{\beta_i\beta_j\beta_n}$. So γ_i and γ_j nummatch iff $(\dagger)_{\beta_i\beta_j\beta_n}$, for $i, j \neq n$.

Finally, consider the nummatching of γ_i with γ_n. The blocks nummatch if and only if

$$a_1.....a_{n-1}.b_{in} = \lambda_n.b_{ni}$$

The left-hand-side of this condition is $b_{1n}...b_{ni}...b_{(n-1)n}.b_{in}$, which, permuting b_{in} with b_{ni}, is $b_{1n}...b_{in}...b_{(n-1)n}.b_{ni}$, which is the right-hand-side. Thus the condition is true, and the blocks nummatch. ∎

Lemma 5.2.4 *The running time required to verify the NMC for a set of blocks having a complete communication graph is quadratic in the number of processes.*

Proof: The total number of ($\dagger$)-conditions to be checked for n-matching is $\binom{n-1}{2} = (n-1)(n-2)/2$. ∎

The ($\dagger$)-condition for any triple of processes in an n-process system can be derived from the $O(n^2)$ ($\dagger$)-conditions given in Theorem 5.2.3. This may be seen by permuting the processes in the statement of the theorem, e.g. in order to show $(\dagger)_{\beta_i\beta_j\beta_k}$, for $i \neq j \neq k \neq i$, define $\alpha_p = \beta_p$ for $p < n, p \neq k$; $\alpha_n = \beta_k$ and $\alpha_k = \beta_n$, and then apply the theorem to the α_i.

The following is a corollary of the proof of theorem 5.2.3.

Lemma 5.2.5 *Process systems whose communication graph is tree-structured are always nummatchable.*

Proof: In any $(\dagger)$-condition, both products are zero, since at least one number on each side is zero. Hence all $(\dagger)$-conditions are trivially satisfied. ▮

5.2.2 Non-Complete Communication Graphs

Suppose $b_{ji} \neq 0$. Define $r_{ij} = \frac{b_{ij}}{b_{ji}}$. If the denominators are non-zero, the condition $(\dagger)_{\beta_i\beta_j\beta_k}$ may also be written in the form $r_{ij} = r_{ik}.r_{kj}$. If $b_{ji} = 0$ but $b_{ij} \neq 0$, or vice versa, then the blocks cannot nummatch, so we assume for the rest of this section that $b_{ji} = 0 \Leftrightarrow b_{ij} = 0$. We call blocks β_i and β_j such that $b_{ji} = b_{ij} = 0$ *non-communicating blocks.* We handle non-complete connected communication graphs by positing a suitable r_{ij} between non-communicating blocks β_i and β_j. Because we talk about paths through the communication graph, we use abbreviated notation $\langle i_1 \ldots i_k \rangle$ to represent a path from β_{i_1} to β_{i_k}.

The Path Condition. Suppose there are two paths $\Pi_1 = \langle c_1 \ldots c_i \rangle$ and $\Pi_2 = \langle d_1 \ldots d_j \rangle$ between $\beta_{c_1} = \beta_{d_1}$ and $\beta_{c_i} = \beta_{d_j}$ in the comm graph. Suppose further that Π_1 and Π_2 are disjoint except for their endpoints, i.e. they are *node-disjoint* paths. The *path condition for* Π_1 *and* Π_2 states that

$$r_{c_1c_2}.r_{c_2c_3}.\ldots.r_{c_{i-1}c_i} = r_{d_1d_2}.r_{d_2d_3}.\ldots.r_{d_{j-1}d_j}$$

The *Path Condition* is the path condition for every pair of node disjoint paths.

The Numerical Matching Condition. The NMC for a non-complete comm graph is the Path Condition, equivalently the Cyclic-Path Condition (defined below). For a complete comm graph, this condition is equivalent to the $O(n^2)$ collection of $(\dagger)$ conditions.

The Cyclic-Path Condition. The path condition may easily be rephrased as a condition on the cycle $\Pi_1{}^\frown\Pi_2{}^{rev}$, where s^{rev} is the reverse of s, and $^\frown$ is concatenation.

Suppose $\Pi_1{}^\frown\Pi_2{}^{rev} = \Pi = \langle c_1 \ldots c_{k+1}\rangle$, where $k+1 = i+j$ and $c_1 = c_{k+1}$. We require that the cycle be *simple*, i.e. it contains no nodes visited twice, and we also relabel the blocks $\beta_{c_1}, \ldots, \beta_{c_k}$ as $\beta_1, \ldots, \beta_k$ for convenience. The *cyclic-path condition for* Π is

$$b_{12}.b_{23}.\ldots.b_{k1} = b_{1k}.b_{k(k-1)}.\ldots.b_{21}$$

The formulation of the condition for general cycles is identical to this, and easily follows from the formulation for simple cycles. The (†) condition is the cyclic-path condition for a 3-cycle.

The *Cyclic-Path Condition* is the cyclic-path condition for every cycle (or simple cycle) Π.

Lemma 5.2.6 (Nummatching Along a Simple Cycle) *Let β_i be the loopbody of $\mathtt{P}_i$. Suppose the communication graph between $\beta_1, \ldots, \beta_k$ is a simple cycle. Then the β_i are nummatchable if and only if the Cyclic-Path Condition is satisfied.*

The proof is in Section 5.3.

Theorem 5.2.7 *Blocks in the same connected component of a communication graph are nummatchable if and only if the path condition holds for any two paths having the same endpoints.*

Proof: The proof follows directly from lemma 5.2.6 and the algorithm used in its proof. ∎

Theorem 5.2.8 (Correctness of NMC) *Assume that the Sequence Condition is satisfied. If the NMC fails, then the collection of loopbodies on which it fails deadlocks.*

The proof is in Section 5.3.

An example of a deadlock condition which may occur despite the Numerical Matching Condition and the Sequence Condition holding is shown in Figure 5.4.

Theorem 5.2.9 *Communicating loopbodies can deadlock even if the NMC and the Sequence Condition hold.*

loop	loop	loop
W(BA)	W(BA)	W(CB)
W(CA)	W(BA)	W(CA)
	W(CB)	W(CA)
endloop	endloop	endloop
ab = 1	**bc = 1**	**ca = 2**
ac = 1	**ba = 2**	**cb = 1**

Figure 5.1: Deadlocking loopbodies that satisfy the NMC

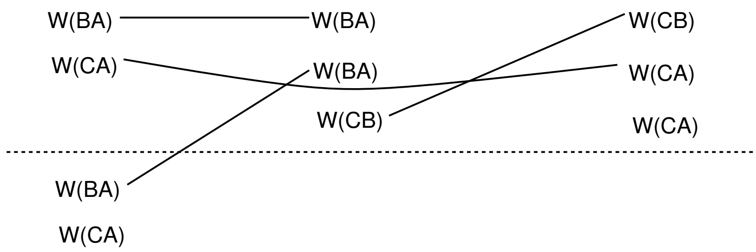


Figure 5.2: The cycle

loop	loop	loop
W(BA)	W(BA)	W(CB)
W(CA)	W(CB)	W(CA)
		W(CA)
endloop	endloop	endloop
ab = 1	**bc = 1**	**ca = 2**
ac = 1	**ba = 1**	**cb = 1**

Figure 5.3: Loopbodies that do not satisfy the NMC

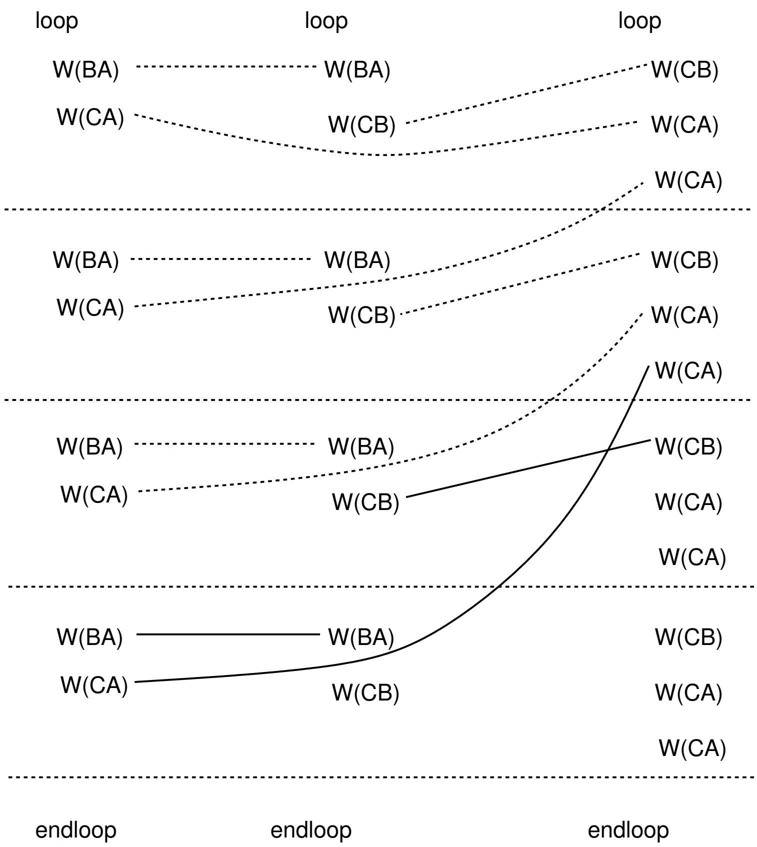


Figure 5.4: The deadlocking duplication

numeric-matching() ::= { If communication graph is not complete
then compute-transitive-closure;
choose an arbitrary node β_n;
for $i < j \neq n$ compute $(\dagger)_{\beta_i\beta_j\beta_n}$;
if any false then return(false) }

compute-transitive-closure ::= { Compute r_{ij} for each edge missing,
using breadth-first-search from each node }

Figure 5.5: Numerical Matching Algorithm

Proof: Follows from the counterexample given in Figures 5.1 and 5.2. Loop `A` must be iterated twice to match the communications in `B` and `C`. The Sequence Condition obviously holds. ∎

The algorithm for checking the NMC for an arbitrary connected communication graph is given in Figure 5.5. It dynamically calculates the r_{ij} using some breadth-first search technique (e.g. a modification of the Warshall algorithm). Note that the running time is a function of the number of processes, which in general is very much smaller than the size of the input.

Theorem 5.2.10 *The NMC can be checked in time quadratic in the number of processes if the communication graph is complete; otherwise, the running time is at worst cubic in the number of processes.*

Proof: The quadratic running time for a complete graph follows from lemma 5.2.4. If the graph is not complete, the transitive closure can be computed in cubic time. ∎

5.3 Proofs

5.3.1 The Sequence Condition

Proof of the Sequence Matching Theorem: We prove Theorem 5.1.1. The theorem stated: *Suppose α and β are blocks such that $r.\alpha = s.\beta + \gamma$, where $\gamma < \beta$, and integers $r, s > 1$. Then there is a sequence σ such that $\alpha = x.\sigma$, $\beta = y.\sigma$, and $\gamma = p.\sigma$, for some integers x, y, p. Furthermore, σ can be taken to be of length gcd(length(α), length(β)).*

Proof: If $\alpha = \beta$, then the theorem trivially holds. So suppose that $\alpha \neq \beta$. Given two blocks α, β with $\beta < \alpha$, we have $\alpha = \beta + \mu$ for some block μ. Define $\rho(\alpha, \beta) = \rho(\beta, \alpha) = \beta$, $\sigma(\alpha, \beta) = \sigma(\beta, \alpha) = \alpha$, and $\tau(\alpha, \beta) = \tau(\beta, \alpha) = \mu$. The functions ρ, σ, τ are defined for only arguments α, β which are comparable, i.e. $\beta < \alpha$, or *vice versa.*

Let $\alpha = k.\beta + \gamma$, where $\gamma < \beta$. (If $\alpha < \beta$, then $k = 0$). If $\gamma = \langle\rangle$, the result follows easily, so assume $\gamma \neq \langle\rangle$. Since by assumption $r > 1$, we know that γ must be a prefix of the $k + 1^{st}$ β. Hence, $\beta = \gamma + \delta$.

Let

$$2r.\alpha = 2(s.\beta + \gamma) = j.(\gamma + \delta) + \gamma'$$

with $\gamma' < \beta$, and $2rk \leq j < 2r(k + 1)$. Also, by the above equation,

$$2r.\alpha = (s.\beta + \gamma) + (s.\beta + \gamma) = (s.(\gamma + \delta) + \gamma) + (s.(\gamma + \delta) + \gamma)$$

$$(s.(\gamma+\delta)+\gamma)+(s.(\gamma+\delta)+\gamma) = j.(\gamma+\delta)+\gamma' = s.(\gamma+\delta)+\gamma+m.(\delta+\gamma)+\delta+\gamma'$$

where $s - 1 \leq m \leq s$. Since $s > 1$ by assumption, we conclude $m > 0$. Removing the sequence $s.(\gamma + \delta) + \gamma$ from the heads of both sides yields

$$s.(\gamma + \delta) + \gamma \;=\; m.(\delta + \gamma) + \delta + \gamma'$$

Comparing the initial sequences of length $|\gamma + \delta|$, we conclude that

$$(\gamma + \delta) = (\delta + \gamma)$$

If $\gamma = \delta$, we're done, with $\beta = 2.\gamma$ and $\alpha = (2k + 1).\gamma$

If $\gamma \neq \delta$ then one sequence has to be shorter than the other, say $|\gamma| < |\delta|$, and we can compare the heads of length $|\gamma|$ in the commuting equation to obtain $\gamma < \delta$, or *mutatis mutandis* $\delta < \gamma$ if $|\delta| < |\gamma|$ instead.

Set $\rho_1 = \rho(\gamma, \delta), \sigma_1 = \sigma(\gamma, \delta)$, and $\tau_1 = \tau(\gamma, \delta)$. Then, rewriting the commuting equation above gives

$$(\rho_1 + \sigma_1) = (\sigma_1 + \rho_1)$$

from which it follows that

$$\rho_1 + (\rho_1 + \tau_1) = (\rho_1 + \tau_1) + \rho_1$$

which, removing the head of length $|\rho_1|$, yields

$$(\rho_1 + \tau_1) = (\tau_1 + \rho_1)$$

More generally, for $k \geq 1$, assume that

$$(\rho_k + \tau_k) = (\tau_k + \rho_k)$$

and $(\rho_k \neq \tau_k)$. We shall show that

$$(\rho_{k+1} + \tau_{k+1}) = (\tau_{k+1} + \rho_{k+1})$$

where

$$\sigma_{k+1} = \sigma(\sigma_k, \tau_k), \rho_{k+1} = \rho(\sigma_k, \tau_k), \tau_{k+1} = \tau(\sigma_k, \tau_k)$$

There are two cases:

$\tau_k < \rho_k$: then $\sigma_{k+1} = \rho_k$ and $\rho_{k+1} = \tau_k$. In this case,

$$(\rho_k + \tau_k) = (\tau_k + \rho_k) = (\sigma_{k+1} + \tau_k) = (\tau_k + \sigma_{k+1}) =$$

$$(\rho_{k+1} + \tau_{k+1}) + \rho_{k+1} = \rho_{k+1} + (\rho_{k+1} + \tau_{k+1})$$

and deleting the sequence of length $|\rho_{k+1}|$ from the head of both gives

$$\tau_{k+1} + \rho_{k+1} = \rho_{k+1} + \tau_{k+1}$$

$\rho_k < \tau_k$: We omit the proof, which is similar to the previous case.

Since $|\sigma_{n+1}| < |\sigma_n|$, by the well foundedness of $<$ on the positive integers, this process must halt, with $\rho_p = \tau_p$ for some positive integer p. We construct two sequences of positive integers a_i and b_i for each i from p down to 1, by 'reverse' induction. Suppose $\rho_p = \tau_p = \nu$. We find a_k and b_k such that $\rho_k = a_k.\nu, \tau_k = b_k.\nu$, for each $p \geq k > 1$. For our two cases,

$\tau_{k-1} < \rho_{k-1}$: then

$$\rho_{k-1} = \sigma_k = (a_k + b_k).\nu$$

and

$$\tau_{k-1} = \rho_k = a_k.\nu$$

So

$$a_{k-1} = (a_k + b_k)$$

and

$$b_{k-1} = a_k$$

$\rho_{k-1} < \tau_{k-1}$: We again omit the proof, since it's similar.

In either case, it is easy to check that

$$\sigma_{k-1} = (a_{k-1} + b_{k-1}).\nu = (2a_k + b_k).\nu$$

and hence

$$(a_{k-1} + b_{k-1}) = (2a_k + b_k)$$

The reverse recursion equations tell us that there are integers a_1 and b_1 such that $\sigma_1 = (a_1 + b_1).\nu$ and $\rho_1 = a_1.\nu$. Thus,

$$\beta = \gamma + \delta = \sigma_1 + \rho_1 = (2a_1 + b_1).\nu$$

and

$$\alpha = k.(2a_1 + b_1).\nu + a_1.\nu = (2a_1k + a_1 + b_1k).\nu$$

if $\gamma < \delta$, and

$$\alpha = k.(2a_1 + b_1).\nu + (a_1 + b_1).\nu = (2a_1k + a_1 + b_1k + b_1).\nu$$

if $\delta < \gamma$. In either case, $\alpha = x.\nu$ for some positive integer x, and $\beta = y.\nu$ for $y = a_1 + b_1$. Finally, it is easy to see that since α and β are composed of identical iterated sequences, then a subsequence of length the greatest common divisor of α and β will suffice for such a common subsequence. ▮

Proof of the Complexity of Checking the SC: Theorem 5.1.4 says that checking the SC for the loopbodies of n communicating processes is linear in the total number N of commstats. We provide the proof.

Let β_i be the loopbody of $\mathtt{P}_i$, n_i the total number of commstats in β_i, and b_{ij} the number of commstats in β_i communicating with β_j. Thus, $N = \Sigma_i n_i$ and $n_i = \Sigma_j b_{ij}$. We show that checking the SC for each pair of blocks β_i and β_j is $O(b_{ij} + b_{ji})$, hence the total check is $O(\Sigma_{i,j}(b_{ij} + b_{ji}))$.

The b_{ij} can all be determined in a single pass over all N commstats. For each processor $\mathtt{P}_i$ a linked list of the names of the other processors with which $\mathtt{P}_i$ communicates is maintained. The list is such that the entry for $\mathtt{P}_j$ occurs before that for $\mathtt{P}_k$ if $j < k$. In addition, the value of b_{ij} is entered into the ijth position of an $n \times n$ matrix $\mathcal{M}(i,j)$. This is done to avoid having to scan entries of $\mathcal{M}$ that do not correspond to a pair of communicating processes. Finally, each entry $\mathcal{M}(i,j)$ that corresponds to a pair of communicating processes contains a pointer to a list that is of length dictated by the sequence matching theorem and that is constructed during the second pass. The SC can also be verified during the same second pass by maintaining a pointer for each such list. ▮

5.3.2 The Numerical Matching Condition

We give here the proofs of the theorems involved in numerical matching.

Proof of Nummatching Along a Simple Cycle: We give the proof of Lemma 5.2.6. First, let Π be the simple cycle between blocks $\beta_1, \ldots, \beta_k$. Suppose the Cyclic-Path Condition is satisfied, so the cyclic-path condition for Π is satisfied. Select a distinguished block, say β_1. By assumption β_1 communicates only with β_2 and β_k. To simplify the proof we first add commstats between all non-communicating blocks $\beta_1, \beta_i, 1 < i < k$, then show that the blocks with the added commstats are nummatchable, and then observe that the blocks without the new commstats are also nummatchable.

We first determine how many commstats to add that involve communications between β_1 and β_3. The matching condition $(\dagger)_{\beta_1\beta_2\beta_3}$, can be stated using r_{ij}'s as $r_{13} = r_{12}.r_{23}$. Since r_{13} is undefined, we add commstats to β_1 and β_3 as follows. Let $r_{12}.r_{23} = x/y$, where $gcd(x, y) = 1$. We add x copies of `W(P`$_1$`,P`$_3$`)` to β_1 and y copies of `W(P`$_1$`,P`$_3$`)` to β_3 (say, at the end of each). The condition $(\dagger)_{\beta_1\beta_2\beta_3}$ is satisfied with these additions.

Suppose we have added commstats to β_1 and β_i, for $1 < i < j$. We now compute the number of commstats to add that involve communications between β_1 and β_j. The matching condition $(\dagger)_{\beta_1\beta_{j-1}\beta_j}$ is: $r_{1j} = r_{1(j-1)}.r_{(j-1)j}$. By assumption $r_{(j-1)j}$ can be computed from the original communications between β_{j-1} and β_j, and we have just computed the value of $r_{1(j-1)}$. Therefore, we can compute the (unique) value of r_{1j} and add the corresponding new commstats to β_1 and β_j.

The computation of the r_js is equivalent to the following: since $r_{13} = r_{12}.r_{23}$, we have $r_{14} = r_{13}.r_{34} = r_{12}.r_{23}.r_{34}$. Iterating up to k gives $r_{1k} = r_{12}.r_{23}.\ldots.r_{(k-1)k}$. By theorem 5.2.3 and its proof, the blocks are nummatchable if and only if this condition is satisfied. But since the new blocks are nummatchable, so are the old blocks with the added commstats removed.

Conversely, suppose blocks $\beta_1, \ldots, \beta_k$ match, and let $m_1, \ldots, m_k$ be the minimum number of duplications required to make all the β_i nummatch. In other words, $m_i.b_{i(i+1)} = m_{i+1}.b_{(i+1)i}$, for $1 \leq i < k$, and $m_1.b_{1k} = m_k.b_{k1}$. Therefore, $m_{i+1}/m_i = b_{i(i+1)}/b_{(i+1)i} = r_{i(i+1)}$. Simple algebra gives $r_{1k} = r_{12}.r_{23}.\ldots.r_{(k-1)k}$, and the result follows immediately. ∎

We now give the proof of Theorem 5.2.8.

Proof of Correctness of NMC: Suppose the NMC fails, i.e. the cyclic-path condition fails on some simple cycle of blocks $\beta_1 \ldots \beta_k\beta_1$. We shall duplicate the blocks, and construct the message flow graph of the duplicated blocks. The message flow graph has directed edges connecting consecutive commacts in the same block, and undirected edges connecting commstats in different blocks that represent the same commact. A (message flow graph) *cycle* is a sequence of nodes $\langle x_1, y_1, x_2, y_2, \ldots, x_j, y_j, x_1\rangle$, where $x_i, y_i \in \beta_{m_i}$ for $1 \leq i \leq j$, such that: $\{y_i, x_{i+1}\}$ for $1 \leq i < j$ and $\{y_j, x_1\}$ are undirected edges; and there is a sequence of directed edges from x_i to y_i in β_{m_i} for $1 \leq i \leq j$. It follows from the correctness proof above of the message flow graph construction algorithm that, under the assumption that the Sequence Condition holds, a collection of blocks deadlocks if and only if either there is an isolated node, or there is a cycle in the message flow graph. We shall construct a cycle in the message flow graph by duplicating the blocks enough times.

Since by lemma 5.2.5 we can match any tree, we can duplicate blocks so that β_i and β_{i+1} nummatch for $1 \leq i < k$. However, β_1 and β_k cannot nummatch after this duplication, since otherwise the NMC holds. Thus, without loss of generality we can assume $b_{i(i+1)} = b_{(i+1)i}$ for $1 \leq i < k$, and $b_{1k} \neq b_{k1}$, say $b_{1k} > b_{k1}$. We also assume without loss of generality that there are no commstats in any β_i that communicate with other processes outside of $\beta_1, \ldots, \beta_k$, so that we can treat this cycle as a closed subsystem, and when we duplicate the blocks there will be no isolated nodes.

We construct a cycle by duplicating the loop bodies 'enough' times. (There may a cycle already in the message flow graph of the first copies of each block, or a cycle may occur during duplication, in which case we would be done already before reaching the cycle we construct.) Let δ_i be the original loopbody of β_i (we reserve the term 'β_i' for the i'th process, which after duplication consists of some integral number of copies of δ_i). We refer to the p'th copy of δ_i in β_i as β_i^p. We say that a U edge e is *in* β_i^p if one end of e is in β_i^p. We write $x > y$, where x and y are nodes in the same process, to mean that x occurs further down in the code than y.

Observe, by our assumption that $b_{i(i+1)} = b_{(i+1)i}$ for $1 \leq i < k$, that there is at least one U-edge $\{y_i^p, x_{i+1}^p\}$ between β_i^p and β_{i+1}^p, for $i < k$ and any $p > 0$. We duplicate the loop bodies enough so that some node x_1 in β_1^p is matched with a node y_k in β_k^q, where $q \geq (p+k)$. Then we may pick a node y_1 in $\beta_1^{(p+1)}$ matched with some x_2 in $\beta_2^{(p+1)}$, some $y_2 > x_2$ in $\beta_2^{(p+2)}$ matched with some x_3 in $\beta_3^{(p+2)}$, ..., some $y_{k-1} > x_{k-1}$ in $\beta_{k-1}^{(p+k-1)}$ matched with some x_k in $\beta_k^{(p+k-1)}$. Hence $y_k > x_k$, y_k is matched with x_1, and $x_1 < y_1$, thus we have a cycle $\langle x_1, y_1, x_2, \ldots, y_{k-1}, x_k, y_k, x_1\rangle$.

Finally, we give an upper bound for how large p must be so that some x_1 in β_1^p matches with some y_k in β_k^q, where $q \geq (p+k)$. By our assumptions, in β_i^1 and β_j^1 all commstats match, except for the 'extra' commstats in β_1^1 communicating with β_k. Let $s = (b_{1k} - b_{k1})$ be the number of these 'extra' commstats, and $t = b_{k1}$, the number of commstats in β_k^1 which communicate with β_1^1. There are thus $s+t$ commstats in β_1^1 communicating with β_k. After $p-1$ duplications of each loopbody, $\beta_i = p.\delta_i$, and there are $p.(s+t)$ commstats in β_1 communicating with β_k and $p.t$ commstats in β_k communicating with β_1. If p is chosen large enough so that $p(s+t) > (p+k)t$ (i.e. $ps > kt$), then at least one commstat, call it x_1, in β_1^p matches with a commstat, call it y_k, in $\beta_k^{p'}$, where $p' \geq p+k+1$, and we are done. ∎

Chapter 6

Traces and Sequentialization

As we noted in Chapter 1, sequentialization of parallel process specifications has many uses. In this chapter, we show how traces may be defined formally, and how sequentializations may be obtained from the mfg and the mfg algorithm. We also show that the mfg algorithm constructs the *minimum* message flow graph (mmfg), that is, an mfg with a minimal number of nodes, and that the system loop process obtained from the mfg algorithm is also minimal. We consider in this chapter only systems of processes for which the mfg algorithm normal-terminates. (In this case, the processes don't deadlock, by Theorem 4.1.6).

Each system loop process corresponds to a sequentialization, but not vice versa. A system loop process executes commacts (or, rather, equivalent uniprocessor statements performing the same task as the commacts when the processes were concurrent) in an eventually periodic order, whereas an arbitrary sequentialization does not have to satisfy such a periodic constraint.

To construct a sequentialization as the mfg algorithm proceeds, uniprocessor equivalent statements are output as corresponding U edges are drawn by the mfg algorithm. The keyword `loop` must be output after the first `while` loop of `main` is completed (all preamble nodes have been matched), and similarly the keyword `endloop` after all loop-body isolated nodes have been processed (all loopbody nodes have been matched). This is a particular example of a system loop process *corresponding* to a message flow graph, as we define below.

We first formally define the trace of a loop process as an ordered set of copies of commstats. From this, we define the collection of commacts. A theorem in Chapter 4 enables us to do this by factoring out by the equivalence relation U that holds between two copies if and only if the nodes which they copy are joined by a U edge in the mfg. The ordering relations on the traces of each loop process induce an ordering relation on the collection of commacts, and the traces of the system are then simply all total orderings of commacts that are consistent with this induced ordering relation. A sequentialization is a finite representation of a trace, and we look at those whose sequence of commacts is eventually periodic, since these are the traces generated by system loop processes. We need to introduce a certain amount of notation, but the basic ideas are straightforward.

6.1 Traces

Let $Q_1, \ldots, Q_n$ be some system of loop processes for which the mfg algorithm normal-terminates, M the mfg, and $P_1, \ldots, P_n$ the process code after the mfg algorithm has been applied. We assert that the unique traces of Q_i and P_i are identical, for $1 \leq i \leq n$, and therefore without loss of generality, and for convenience, we use only the P_i in the definitions that follow. Variables x, y ranging over commstats should be thought of as ranging over occurrences of statements (i.e. statement labels, e.g. the fourth commstat in the preamble of P_k), rather than over types of statement (e.g. whether this commstat is a `W(AB)` or a `W(CA)`, etc.).

We define the relation $\sim$ to hold of two commstats x and y if and only if they are in the same process, namely $x \sim y$ for $x, y \in M$ iff $P_x = P_y$ (recall that P_x is the process to which x belongs). We define the predicate $Ab(x, y)$ to hold iff $x \sim y$ and x is above y. It is easy to see that Ab is a strict linear ordering of the commstats in each process (i.e. each $\sim$-equivalence class), and that, considered as a set of pairs, $Ab \subset \sim$. We have previously defined a system trace informally as any execution of the system that is consistent with the ordering of the commstats in each process. We now give a more formal definition, by defining a structure *trace(P_k)* to represent the trace generated by a process P_k (recall that each individual loop process generates a unique trace, although the concurrent execution of the system of loop processes may have many traces). We then use the set of *trace(P_k)*'s to define the set of system traces.

6.1.1 The Trace of a Loop Process

Defining the Field of $trace(P_k)$**.** For each node x in the loopbody of $P_k, k \leq n$, we define an infinite sequence of nodes $x^1, x^2, \ldots, x^i, \ldots, i \in \omega$ (ω

is the natural numbers) and appropriate relations between them, to represent the occurrences of x in the iterations of the loopbodies in a trace of P_k. To simplify notation, we also write a commstat x in `preamble`(M) as x^1. For example, if P_k is `(a; b; loop c; d; e; endloop)`, then we obtain the nodes $\mathtt{a}^1$, $\mathtt{b}^1$, $\mathtt{c}^1$, $\mathtt{d}^1$, $\mathtt{e}^1$, $\mathtt{c}^2$, $\mathtt{d}^2$, $\mathtt{e}^2$, $\mathtt{c}^3$, $\mathtt{d}^3$, $\mathtt{e}^3$, ... We extend $\sim$ to the x^i in the obvious way: $x^i \sim y^j$ iff $x \sim y$.

Intuitively, we want a superscript i on a node x^i in this sequence to indicate that this node corresponds to the i'th time that x is executed (this happens, since by our assumption there is no deadlock). More precisely, we want the list of nodes of the form x^i to correspond one-to-one with the commacts in the unique execution trace of P_k, in such a way that if x^i corresponds with a given commact, then that commact occurs by means of the i'th execution of x in P_k. We formalize this by defining an ordered set *trace(P_k)* which represents the trace of P_k. The field of *trace(P_k)* is the set

$$\mathcal{T}(P_k) = \{x^1 \mid x \in \text{preamble}(P_k)\} \cup \{x^i \mid x \in \text{loopbody}(P_k) \ \wedge \ 1 \leq i \in \omega\}$$

Define also $\mathcal{T}_S = \bigcup_{1 \leq k \leq n} \mathcal{T}(P_k)$. The sets $\mathcal{T}(P_k)$ are just the $\sim$-equivalence classes of $\mathcal{T}_S$.

Defining the Ordering on ***trace***(P_k)**.** We can extend the *'Above'* relation Ab on commstats in the P_k's to a relation $Ab^\dagger$ on $\mathcal{T}_S$ in the obvious way. Define $Ab^\dagger(x^i, y^j)$ iff $x \sim y$, and either $i < j$, or $i = j$ and $Ab(x, y)$. It is immediate that $Ab^\dagger$ is a strict linear order on each $\sim$-equivalence class $\mathcal{T}(P_k)$. Finally, let $Ab_k^\dagger$ be $Ab^\dagger$ restricted to the domain $\mathcal{T}(P_k)$. We define *trace*(P_k) to be the ordered set $\langle \mathcal{T}(P_k), Ab_k^\dagger \rangle$.

Representing the Trace. We need formally to show how *trace*(P_k) represents the execution trace of P_k. First, observe that the order type of *trace*(P_k) is ω (the order type of the natural numbers). The order type of the trace of P_k, the sequence of commacts engaged in by P_k as it executes, is also ω. Thus, there is a unique order isomorphism f_k between *trace*(P_k) and the trace of P_k which maps the i'th element in the order on *trace*(P_k) onto the i'th commact in the trace. This order isomorphism is a function f_k from *trace*(P_k) to the trace of P_k such that: (a) for every $x \in P_k$ and every $i, 1 \leq i \in \omega$, $f_k(x^i)$ is a commact engaged in by P_k; (b) for any commact A engaged in by P_k, there is an $x \in P_k$ and an $i, 1 \leq i \in \omega$ such that $A = f_k(x^i)$; and finally (c) $Ab^\dagger(x^i, y^j)$ if and only if $f_k(x^i)$ occurs before $f_k(y^j)$ in the unique trace of P_k. There is such an f_k for each $k, 1 \leq k \leq n$. Since the nodes in different processes are disjoint, the domains of these f_k's are disjoint. So we may define the function $f = \bigcup_{1 \leq k \leq n} f_k$ (this is simply a formal way of dropping the subscripts from the f_k's without fear of ambiguity.) From the previous considerations, the lemma below easily follows:

Lemma 6.1.1 *Let $P_1, \ldots, P_n$ be loop processes in the mfg M. For any process P_k, any $x \in P_k$, and any $1 \leq i \in \omega$, the commact $f(x^i)$ occurs as a result of executing statement x for the i'th time in P_k.*

6.1.2 From Commstats to Commacts

While we could try to use the function f to pass from nodes in $\mathcal{T}_S$ to commacts, since two nodes x^i, y^j participate in the same commact if and only if $f(x^i) = f(y^j)$, it is not obvious how to construct the function f. By Theorem 4.1.4, the U edges in the mfg indicate precisely which mfg nodes participate in the same commact. We use this observation to define a relation $U(x,y)$ such that $U(x^i, y^j)$ if and only if $f(x^i) = f(y^j)$.

Define $U(x^i, y^j)$ iff $i = j$ and there is a U edge between x and y in M. It is immediate that U is an equivalence relation, and that each equivalence class of U has exactly two elements. So we may pass from considering a node x in $\mathcal{T}_S$ to considering the equivalence class x/U of x in U (formally, $x/U = \{y \mid U(x,y)\}$). This equivalence class may be identified with the commact executed by x. Notice also that the equivalence class is a pair set, as is a U edge in the mfg. Thus the U edge containing x is the same set as x/U.

By Theorem 4.1.4, any commact occurs by means of the simultaneous execution of two commstats x, y such that $U(x,y)$. Furthermore, by the same theorem it follows that every previous execution of x has been accompanied by a simultaneous execution of y. Thus, x and y have each executed exactly the same number of times at any point in a trace, and we obtain the following lemma:

Lemma 6.1.2 *Let $P_1, \ldots, P_n$ be loop processes in the mfg M. Let $P_x \neq P_y$. Then x^i, y^j participate in the same commact, that is $f(x^i) = f(y^j)$, if and only if $i = j$ and $U(x^i, y^j)$.*

Proof: The lemma follows by an easy induction on i, using Theorem 4.1.4 and the definitions of U and f. ∎

Extending the Ordering on $\mathcal{T}_S$ to an Ordering on Commacts. The relation $Ab^\dagger$ has domain $\mathcal{T}_S$. Define the set $\mathcal{T}_A = \{x/U \mid x \in \mathcal{T}_S\}$. Define the relation $<_{Ab}$ on $\mathcal{T}_A$ by the following: (i) $(x^i/U <' y^j/U)$ iff $Ab^\dagger(x^i, y^j)$; (ii) $<_{Ab}= (<')^*$ (i.e. $<_{Ab}$ is the transitive closure of $<'$).

Lemma 6.1.3 *Let $P_1, \ldots, P_n$ be loop processes in the mfg M. (a) $<_{Ab}$ is a strict partial order; (b) $<_{Ab}$ is discrete, that is for every pair of elements a, b in $\mathcal{T}_A$ such that $a <_{Ab} b$, there is c such that $a \leq_{Ab} c <_{Ab} b$ and for all d, $\neg(c <_{Ab} d <_{Ab} b)$, and also c' such that $a <_{Ab} c' \leq_{Ab} b$ and for all d, $\neg(a <_{Ab} d <_{Ab} c')$; (c) $<_{Ab}$ is well-founded; (d) $<_{Ab}$ is the smallest partial order $<$ satisfying the condition that $(x^i/U < y^j/U)$ if $Ab^\dagger(x^i, y^j)$.*

Proof: (a) and (b) are straightforward. To show (c), we note that for fixed i, $\{x^i \mid x \in M\}$ is finite, and ω is well-ordered, $\mathcal{T}_S$ is well-founded under $Ab^\dagger$ and hence so is $\mathcal{T}_A$ under $<_{Ab}$. For (d), we note that by (a) $<_{Ab}$ is a strict partial order containing the set $\{\langle x^i/U, y^j/U\rangle \mid Ab^\dagger(x^i, y^j)\}$, and that any partial order $<$ containing that set must also be transitive. Therefore $<_{Ab}\subseteq<$. ∎

6.1.3 System Traces

In Chapter 1, Section 1.4.1, we defined a trace of a system of processes to be a sequence of interleaved commacts that respects the ordering of the commacts defined by the order of the commstats in each individual process, and respects the semantics of the communication constructs. So far, we have defined the sequence of commacts of each individual process P_k as *trace*(P_k), and from those the set of commacts $\mathcal{T}_A$ of the system. Further we have defined $<_{Ab}$, the smallest partial order on $\mathcal{T}_A$ consistent with the order on *trace*(P_k) for $1 \leq k \leq n$ by Lemma 6.1.3. Hence, if some total order is consistent with the orders on *trace*(P_k) for $1 \leq k \leq n$, it extends $<_{Ab}$. The corollary to the following lemma shows the converse. We may thus conclude from Lemma 6.1.3 and Corollary 6.1.5 that 'extending $<_{Ab}$' is precisely what it means for a trace to be 'consistent with each individual loop process'.

Lemma 6.1.4 *Let $P_1, \ldots, P_n$ be loop processes in the mfg M. Let $<$ be a total ordering of $\mathcal{T}_A$, and $(x^i/U <_{Ab} y^j/U)$ but $y^j/U < x^i/U$. Then there is some process P_k in which $Ab_k^\dagger(z^l, w^m)$ but $w^m/U < z^l/U$.*

Proof: Let $<$, x^i and y^j be as in the hypothesis of the lemma. By standard results on transitive closure, if $(x^i/U <_{Ab} y^j/U)$ then there is a sequence $x^i = x_0, x_1, \ldots x_p = y^j$ of elements of $\mathcal{T}_S$ such that one of four cases holds: (i) $Ab^\dagger(x_0, x_1), U(x_1, x_2), Ab^\dagger(x_2, x_3), \ldots, U(x_{p-2}, x_{p-1}), Ab^\dagger(x_{p-1}, x_p)$; (ii)-(iv) are similar, where the sequence may instead end with a U relation, or start with a U relation and end with either an Ab or a U relation.

We reason about case (i) only, since the arguments for (ii)-(iv) are similar. Observe firstly that p is an odd integer. Consider the sequence $x^i/U = x_0/U, x_1/U, \ldots x_p/U = y^j/U$. $U(x_1, x_2)$, hence $x_1/U = x_2/U$, and similarly $x_{2i-1}/U = x_{2i}/U$ for $i \leq \lfloor p/2 \rfloor$. Let $s = (p+1)/2$ and define $y_i = x_{2i}$ for $i < s$, $y_s = y^j$. Consider the sequence $x^i/U = x_0/U = y_0/U, y_1/U, \ldots, y_s/U = y^j/U$. By condition (i) above, $Ab^\dagger(y_t, y_{t+1})$ for all $t < s$. Since the y_t/U are all distinct by choice, and $<$ is a total order on $\mathcal{T}_A$, $(y_t/U < y_{t+1}/U)$ or $(y_{t+1}/U < y_t/U)$, for all $t < s$. However, if $(y_t/U < y_{t+1}/U)$ for all $t < s$, then it would follow by transitivity that $y_0/U < y_s/U$, i.e. $x^i/U < y^j/U$, which contradicts the hypothesis.

Hence $(y_{t+1}/U < y_t/U)$ for some $t < s$, so $(x_{2t+1}/U = x_{2t+2}/U = y_{t+1}/U < y_t/U = x_{2t}/U)$, but $Ab^\dagger(x_{2t}, x_{2t+1})$. Pick $z_l = x_{2t}$ and $w^m = x_{2t+1}$ and we are done. ∎

Corollary 6.1.5 *Let $P_1, \ldots, P_n$ be loop processes in the mfg M. Let $<$ be a total ordering of $\mathcal{T}_A$ such that for every $x, y \in \mathcal{T}(P_k)$ such that $Ab^\dagger(x, y)$, $x/U < y/U$. Then $<_{Ab} \subseteq <$.*

By Corollary 6.1.5, we may now formally define a *system trace* as any total order T with field $\mathcal{T}_A$ which extends the relation $<_{Ab}$. The set of system traces is

$$\mathit{Traces} = \{T \mid T \mathit{\ is\ a\ total\ order} \wedge \mathit{field}(T) = \mathcal{T}_A \wedge (<_{Ab} \subseteq T)\}$$

Lemma 6.1.6 *Let $P_1, \ldots, P_n$ be loop processes in the mfg M. If $T \in$ Traces then T is a well-ordering of order type ω.*

Proof: T is a linear ordering extending the well-founded ordering $<_{Ab}$, and therefore T is a well-ordering. Furthermore, for each $y \in T$, $\{x \in T \mid x$ occurs before y in $T\}$ is finite. Hence T yields a well-ordering of order-type ω. ∎

Thus a trace T may be represented by some infinite string $\langle x_i \mid i \in \omega \rangle$ where x_i occurs before x_j in T iff $i < j$, and where $\{x_i \mid i \in \omega\} = \mathcal{T}_A$.

6.1.4 Summary

We summarize the above in a theorem.

Theorem 6.1.7 *Suppose the mfg algorithm normal-terminates on some input, with mfg M. The message flow graph M constructively defines a strict partial order $<_{Ab}$ on commacts, such that the traces of the system are precisely the total orders T such that $field(T) = field(<_{Ab})$ and $<_{Ab} \subseteq T$.*

6.2 Sequentializations

$\mathcal{T}_A$ is infinite. A trace is a total order with field $\mathcal{T}_A$ and therefore is also an infinite object. A sequentialization is a finite representation of a trace. We particularly care about sequentializations given by a pair of finite sequences X, Y such that $T = X^\frown Y^\frown Y^\frown Y^\frown \ldots$ is a trace. These are precisely the sequentializations given by a system loop process, which has the form (X `loop` Y `endloop`).

As noted earlier, when we generate a system loop process from an mfg, we must replace each pair of matched commstats by an equivalent uniprocessor statement. For example, suppose for each variable x in P_i we have an equivalent uniprocessor variable $i.x$. Then the name spaces of all the processors are disjoint in the uniprocessor store. Suppose a commstat in P_i writes the value of P_i's variable s to P_j and its matched commstat in P_j reads the value from P_i and stores it in t, then the uniprocessor equivalent in the system loop process is the assignment $j.t \leftarrow i.s$. However, for convenience in this section, we represent these uniprocessor equivalent statements notationally simply as elements of $\mathcal{T}_A$.

We say that a system loop process (X `loop` Y `endloop`) *corresponds to* an mfg M iff

(a) there is a one-to-one correspondence $f = f_1 \cup f_2$ where f_1, f_2 are both one-to-one correspondences:

$$f_1 : \{x/U \mid x \in \text{ preambles}(M)\} \to X,$$

$$f_2 : \{y/U \mid x \in \text{ loopbodies}(M)\} \to Y;$$

(b) and for $x, y \in \mathcal{T}_A$, if $f(x)$ precedes $f(y)$ in the system loop process then $\neg(y <_{Ab} x)$.

The following lemma follows from results in Section 4.2. As noted in Section 6.1.2, the U edge containing a given commstat x is literally the same set as x/U, and therefore we may compare U edges with equivalence classes freely using the ordering relations defined on either.

Lemma 6.2.1 *If the mfg algorithm terminates normally, then the ordering of the U edges as they are output by the algorithm gives a sequentialization that corresponds to the mfg.*

Proof: The order of output of the U edges from the mfg algorithm defines a total order $\ll$ on $\{x/U \mid x \in M\}$. We need to show that this order $\ll$ extends to a trace (a total order on $\mathcal{T}_A$) $T_{\ll}$, by the periodic repetition of the loopbody indicated above. It suffices to show that $\ll$ is consistent with $<_{Ab}$ restricted to M. $\ll$ is inconsistent with $<_{Ab}$ iff there exist nodes $x, y \in M$ such that $x/U \ll y/U$ and $y/U <_{Ab} x/U$. Call such nodes x, y a *contrary pair*. We show there can exist no contrary pair. Firstly, if u is a topmost isolated node selected for processing in `main`, then there can be no U edge e such that $e \ll u/U$ (i.e. e was previously drawn by the algorithm) and $u/U <_{Ab} e$; since by an argument similar to that in the proof of Lemma 6.1.4, u would not have been topmost isolated when it was selected for processing in `main`, contrary to assumption. Secondly, it follows by induction on the depth of the recursive calls to `ptin` that if $x_0 \ll \ldots \ll x_k$ is the sequence of U edges drawn by a call to `ptin` from `main`, then $x_0 <_{Ab} x_1 <_{Ab} \ldots <_{Ab} x_k$. It follows that at no point does the algorithm handle any node that may be part of a contrary pair, and since all nodes in the mfg are eventually visited, there can thus be no contrary pair.

Let preambles$(M)/U = \{x_0, \ldots, x_i\}$ and loopbodies$(M)/U = \{y_0, \ldots, y_j\}$, where $k < k' \leq i \Rightarrow x_k \ll x_{k'}$, $l < l' \leq j \Rightarrow y_l \ll y_{l'}$. Then the desired sequentialization is `(`$x_0; \ldots; x_i$ `loop` $y_0; \ldots; y_j$ `endloop)`. It is straightforward to show that this sequentialization corresponds to the mfg M. ∎

6.3 Minimality

We justify our claims that the message flow graph algorithm generates a minimum message flow graph, and the sequentialization produced by the mfg algorithm gives a minimum system loop process.

Theorem 6.3.1 *The mfg algorithm constructs a minimum message flow graph if it normal-terminates.*

Proof: Suppose that M is the mfg produced by the algorithm, and M' another mfg. Let P_j^M be the process in M corresponding to the input process P_j , and similarly for $P_j^{M'}$. Suppose for a contradiction that there is a node $x \in$ `preamble(`M`)` such that $x \notin$ `preamble(`M'`)`, and WLOG let x be the first such node that occurred as a parameter to either `ptin`, `unroll`, `match` or `handle` as M was built. Let x' be the node in M' corresponding to x (i.e. if x is the q'th statement executed in a trace of P_x^M, then let x' be the q'th statement executed in a trace of $P_{x'}^{M'}$). By hypothesis, $x' \in$ `loopbody(`M'`)`. Since P_j^M and $P_j^{M'}$ have the same trace (the same as P_j), it follows that `preamble(`$P_{x'}^{M'}$`)`, considered as a sequence of nodes, is a prefix of `preamble(`P_x^M`)`. Also, the processmates y of x and y' of x' are in the preamble of M, respectively the loopbody of M'.

We claim that $x \notin$ `preamble(input)`, since all nodes in `preamble(input)` occur in the preamble of any mfg, in particular M', which contradicts our hypothesis on x. Hence x is a node that was unrolled. This happened by a call to `unroll(`z`)` in `ptin`, where $z = x$ or $x <_{Ab} z$. Now, if $x <_{Ab} z$, $z \notin$ `preamble(`M'`)`, and, further, x has not been 'visited' (occurred as a parameter to any of the four subroutines) at the time that `unroll(`z`)` is called. Hence $z = x$, and `unroll(`x`)` was performed. This was performed during a call therefore to `ptin(`y`)`. Thus y was 'visited' before x. However, $y \in$ `preamble(`M`)`, since it's connected by a U-edge to x and M is an mfg, and $y' \in$ `loopbody(`M'`)`, since y' is connected by a U-edge to x' and M' is an mfg. Hence y is a node in `preamble(`M`)` that is not in `preamble(`M'`)`, and it is 'visited' before x. This contradicts our hypothesis that x was the first such node 'visited'. Hence there cannot be any node $x \in$ `preamble(`M`)` such that $x \notin$ `preamble(`M'`)`, i.e. all nodes in `preamble(`M`)` are in `preamble(`M'`)`.

A similar argument shows that all nodes in `loopbody(`M`)` are also in `loopbody(`M'`)`. We conclude that M is minimal. ∎

Let `A` be the system preamble produced by (the modified) `main` and `B` be the system loop body.

Theorem 6.3.2 *The code* `(A; loop B endloop)` *is a minimal system loop process for the processes* $\mathtt{P}_1, \ldots, \mathtt{P}_n$.

Proof: Let `C` be `(A; loop B endloop)`. Let M be the mfg constructed by the mfg algorithm. `C` by construction has commstats in one-to-one correspondence with U edges of M. This verifies condition (a) of the definition of correspondence. Verifying condition (b) is equally straightforward.

Finally, minimality. Since by Theorem 6.3.1 the mfg algorithm constructs a minimum mfg, no mfg may have fewer nodes than M. Since there are a minimum number of such edges in M, and commstats of `C` are in one-to-one correspondence with edges of M, no system loop process may have fewer commstats than `C`. ∎

We show that sequentialization is angelic for loop processes using convenient primitives; that is, if any execution trace deadlocks, then all execution traces deadlock.

Theorem 6.3.3 *If a system of communicating processes which use convenient primitives has a unique message flow graph, then sequentialization of that system is angelic.*

Proof: All execution traces are consistent with $<_{Ab}$. One of the following is true:

1. the message flow graph algorithm finds an isolated node,
2. the message flow graph algorithm builds a cycle,
3. the message flow graph algorithm finds neither a cycle nor an isolated node.

Since all sequentializations are consistent with the ordering of the message flow graph, the following respective consequences are immediate.

1. all sequentializations have an isolated node, and consequently deadlock,
2. all sequentializations have a cycle, and consequently deadlock,
3. no sequentializations have either an isolated node or a cycle, and consequently none deadlocks.

∎

The following corollary is immediate from Theorem 6.3.3.

Corollary 6.3.4 *Sequentializations of systems of loop processes using convenient primitives are angelic.*

Chapter 7

Asynchronous Communication

The message flow graph algorithm has been stated and verified for convenient primitives. We discuss how to modify the algorithm to process *asynchronous* primitives (which correspond to unbounded channel memory) and *partially asynchronous* primitives (corresponding to finite channel memory). We handle the issues concerning asynchrony informally, indicating how the algorithm must be modified, and providing an intuitive feel for how the modifications work. Since the modifications to the algorithm are introduced informally, arguments that the modified algorithm is correct necessarily inherit at least the same degree of informality.

We need to distinguish communication *events* (what occurs when at least one process executes a commstat), or *commevents*, from commacts (a successfully completed communication action, a sent-from-A-and-received-by-B). For synchronous communication, commevents and commacts are the same thing. There is just one commevent, or commact, involved in completing the communication successfully. With asynchronous communication involving two processes, the commact is effected by potentially two commevents, the send from the originator, and the possibly later receive in the process partner (which could also be simultaneous). Thus, we represent a commact for two-process asynchrony as a ordered pair $<$ send, receive $>$ of commevents such that receive may occur simultaneously with, or later than, send. We shall refer to the first argument generically as a send and the second as a receive. During the time after a send and before a receive, the message is often considered to be resident in a *buffer*, a temporary store than is often considered (and implemented) to behave like a queue. Since channel buffers could fill or overflow during a computation, there may be compiler-generated failure-modes for asynchronous communication that are absent for synchronous communication.

The most commonly studied properties of (pure) asynchrony are that sends are non-blocking, and receives block only if all previous sends have already been received (i.e. the corresponding commacts have executed). A channel behaves as though there were an unbounded buffer, filled by sends and emptied by receives, which never overflows, never loses or duplicates messages, never fails to accommodate a send, or to accommodate a receive when there are incomplete commacts. A receive removes a message from the head of the buffer, and blocks only when the buffer contains no messages.

Buffers in practice are - usually - not unbounded. We need a slightly different construct to deal with communication through bounded buffers. A partially asynchronous commact is also represented by a pair of commevents $<$ send, receive $>$, as for pure asynchrony [AS83]. A receive behaves as in pure asynchrony, but a send may or may not block. Whether a send blocks depends on whether the buffer has spare capacity or not. (We don't consider the case in which existing messages in transit may be corrupted or lost, and therefore we don't consider circumstances such as a newly-generated message overwriting a previously-generated one in the buffer.) A channel **C** behaves as if there is a buffer of fixed finite size cap:**C**. (When there is a single channel $\mathtt{A} \rightarrow \mathtt{B}$, we write cap:AB.)

Buffers have a size, and we can also consider messages to have a size. The commevents of sending and receiving a particular message inherit the size parameter of the message. Hence we could refer both to size(m) where m is a message, and size(y) where y is a commevent. Our interest is in static analysis, and therefore we consider different ways in which a message could have a size which is known at compile-time. A size parameter which arbitrarily varies on performance of a commact does not admit a static analysis, since it is not in general possible to tell at compile time what this size will be on the q'th performance of a particular commstat in the source. Therefore, we consider only size parameters which correlate with the comm*stat* that is executed. There are two types of message size parameter we consider.

- the buffer capacity is measured in terms of the number of messages it may contain, regardless of the length of the messages. This case pertains when messages are signals, or have one standard length. (This is equivalent to letting all messages have size 1.) We refer to the buffers as *message-capacity-limited.* This is a special case of the case below, but is sufficiently common to warrant giving it special attention.

- (the more general case) message sizes may vary, but depend only on which comm*stats* are executed, rather than which comm*events*, and may not vary from execution to execution of a particular commstat (as in the body of a loop). Let m and m' be messages. Given commstat x, say a `W(AB)`, then the requirement is that if both m and m' are transmitted by x, $\mathsf{size}(m) = \mathsf{size}(m')$. (This justifies our use below of the terminology $\mathsf{size}(y)$.) This case pertains when, for example, source code must contain explicit declarations of message size. In this case, message sizes play the part of a type declaration, and commstats are typed by this size. A common case of this is when messages may be either *long* or *short.* We refer to these buffers as *byte-capacity-limited.*

Define a commact to be *incomplete* at a given point in a trace if its send has been executed at that point but its receive has not been executed. A send blocks precisely when

- (for message-capacity limited buffers) there are $\mathsf{cap}:\mathbf{C}$ incomplete commacts, i.e. the number of sends minus the number of receives at this stage in the execution equals the capacity of the buffer. The send unblocks (i.e. becomes available for execution) as soon as there are fewer than $\mathsf{cap}:\mathbf{C}$ incomplete commacts on $\mathbf{C}$ (i.e. a receive is executed on $\mathbf{C}$, rendering an incomplete commact complete);

- (for byte-capacity limited buffers) the size of the message, $\mathsf{size}(m)$, is greater than the remaining capacity of the buffer. The send unblocks precisely when enough receive commevents are executed that the total available unused space in the buffer is greater than or equal to $\mathsf{size}(m)$.

The different communication types may be modelled as in Figures 7.1 through 7.4, in which specific sequences of actions representing the behaviors of the primitives are written as a single atomic action using the construct `begin atomic ... end atomic`. Clearly, one needn't implement the primitives as in the figures; the figures are simply one possible interpretation of the semantics of the primitives in this monograph. (The two different limitations of partial synchrony will be explained below.)

The models use an await construct, which is intended to have the following intuitive semantics. When an await is tested within an atomic action and

```
W(AB) in A:                begin atomic
                               ready-to-write-on-AB
                               await B is ready-to-read-on-AB
                               send-to-B
                           end atomic


W(AB) in B:                begin atomic
                               ready-to-read-on-AB
                               await A is ready-to-write-on-AB
                               receive-from-A
                           end atomic
```

Figure 7.1: Synchronous two-process transparent primitives

```
W(AB) in A:                begin atomic
                               send-to-B
                               b := b+1
                           end atomic


W(AB) in B:                begin atomic
                               await b > 0
                               receive-from-A
                               b := b-1
                           end atomic
```

Figure 7.2: Pure Asynchronous primitives

```
W(AB) in A:              begin atomic
                             await ab < cap:AB
                             send-to-B
                             ab := ab+1
                         end atomic


W(AB) in B:              begin atomic
                             await ab > 0
                             receive-from-A
                             ab := ab-1
                         end atomic
```

Figure 7.3: Primitives for message-capacity-limited buffers

```
W(AB) in A:              begin atomic
                             await (ab+size(m)) <= cap:AB
                             send-to-B(m)
                             ab := ab+size(m)
                         end atomic


W(AB) in B:              begin atomic
                             await ab > 0
                             receive-from-A(m)
                             ab := ab-size(m)
                         end atomic
```

(`size(m)` is a parameter carried 'with' a message `m`.)

Figure 7.4: Primitives for byte-capacity-limited buffers

the predicate is untrue, then the process waits *outside* the atomic block, so that other processes may execute enabled actions in the meantime. When the predicate becomes true, then the atomic action is enabled with control at the statement immediately following the await until the predicate becomes false again, in which case the action reverts to awaiting. There are well-known subtleties involved in implementing such a construct in general (for example, ensuring that the predicate doesn't become false again between the time it is tested true and the time the atomic block is executed) but these problems do not arise for our examples, in which it's easy to see that: (a) a process using an await has only one thread of control; (b) the truth of the predicate depends solely on the state of one channel controlled by two processes; and (c) if process A is awaiting B, then once the specific predicate becomes true, it remains true until A has executed the atomic action that is awaiting. We are not solving general concurrency problems, and consequently choose to use a construct that is intuitively clear for our examples, and not one necessarily applicable to arbitrary predicates.

For partially asynchronous communication, some knowledge about the size and number of incomplete commacts is required for successful execution; this information is used to modify the message flow graph algorithm.

7.1 Modelling with the message flow graph

We model pure asynchronous and partially asynchronous primitives in the message flow graph by changing the U edges from undirected to directed edges, where the direction is from a send to a receive. Deadlock occurs whenever there is a cycle in the partially constructed message flow graph. Since U edges are now directed, they may be traversed only in one direction. So, a configuration which may be a cycle if the U edges are undirected may not be a cycle if the U edges are directed. Such a configuration corresponds to a system that deadlocks with synchronous primitives and does not deadlock with pure asynchronous primitives. Figure 7.5 is an example of two 2-process configurations which both deadlock with synchronous primitives, one of which does not deadlock with pure asynchronous communication. Additionally, partially asynchronous communications may deadlock when the buffer is full.

The following definitions of different types of cycles do not depend on whether or not the commstats involved *actually are* synchronous or asynchronous.

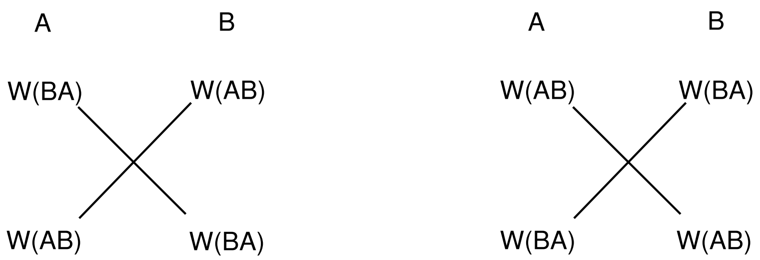


Figure 7.5: Deadlocking and non-deadlocking asynchronous communications.

- A *synchronous* cycle is a sequence of edges that form a cycle if all the interprocess communication edges are taken as undirected.
- An *asynchronous* cycle is a sequence of edges that form a cycle if all interprocess communication edges are taken as directed.

Lemma 7.1.1 *In an asynchronous cycle the topmost cycle node in a process is a* receive*, and the lower node is a* send*.*

Lemma 7.1.1 follows directly from the definition of asynchronous cycle. Thus, synchronous cycles with receives top in all processes that participate in the cycle are also asynchronous cycles.

7.2 Pure Asynchronous Communication

The modification to the algorithm to work with pure asynchronous communication is straightforward. When the algorithm looks at a topmost isolated node x in process `A`, and this node is a `W(AB)`, then a directed edge is drawn immediately from that `W(AB)` to y, its matched `W(AB)` in `B`, and x's successor node in `A` becomes the topmost isolated node in `A`. If y is not the topmost isolated node in `B`, we say that y is now *pending*. If y is the topmost isolated node in `B`, then the next isolated, non-pending, node in `B` becomes topmost isolated. Consequently, `first-node(`α`,P)` must be modified so that

$z \leftarrow$ `the first isolated non-pending node of type` α

(We shall define 'pending' in a slightly different manner when we deal with partial asynchrony.) If, however, no such edge has been drawn to the topmost isolated node in `A`, then the algorithm proceeds as in the synchronous case. If the algorithm ever detects a cycle, then it is straightforward to show that, using the above procedure, all the topmost isolated nodes participating in that cycle are equivalent to receives.

A mixture of both synchronous and asynchronous primitives may be used in the definition of the same system of loop processes. The system will be analyzable using our algorithm, with the adaptation for asynchronous commstats, and the original procedure for synchronous commstats (with the proviso that the `ptin-handle` subroutine call stack for synchronous primitives is popped when a U edge is drawn for an asynchronous primitive). Our definition of *cycle* applies equally to mixtures of both communication types.

7.3 Partially Asynchronous Communication

We now analyze the case in which a buffer of finite message capacity is associated with each message type. For example, for the case in which the only message types involving a pair of processes `A` and `B` is either `W(AB)` or `W(BA)`, there is a finite buffer associated with the `W(AB)` and a different finite buffer associated with `W(BA)`. The two buffers need not have the same capacity.

Synchronous communication may also be thought of as partially asynchronous communication with buffer sizes all identically 0. If a deadlock cycle occurs in the partially asynchronous case, it occurs with all buffer sizes ≥ 0. The cycle must also be a deadlock cycle in the case in which less buffer capacity is available, in particular when buffers all have 0 capacity available. It follows from this observation that deadlock cycles with partially asynchronous communication are synchronous cycles. It also follows that asynchronous cycles are also deadlock cycles for partially asynchronous communication, since if there is a deadlock cycle with buffers of unbounded capacity, there is still one when the buffers have bounded capacity. We conclude that all asynchronous cycles are partially asynchronous cycles, and all of these are synchronous cycles.

We use a data structure that records the total size of messages in a buffer, as well as the capacity of that buffer (also in terms of size of messages). This can be implemented using a collection of tuples $\langle Q, \mathtt{cap}{:}Q, q\rangle$ where Q is the buffer (or channel) name, $\mathtt{cap}{:}Q$ is the capacity for Q, and q is the currently used capacity of Q. For each buffer Q between, say, `A` and `B`, q is initialized to 0. Let `W(AB)` be associated with the tuple $\langle AB, \mathtt{cap}{:}AB, ab\rangle$.

As we have noted, the behavior of finite buffers lies between that of the pure asynchronous and the synchronous cases. The system behaves as in the pure asynchronous case as long as the appropriate buffers have some spare capacity. In terms of our algorithm, if the topmost isolated node in `A` is a `W(AB)`, and if AB is able to receive the message (for message-capacity-limited buffers, ($ab <$ `cap`$:AB$); for byte-capacity-limited buffers, size$(m) \leq$ (`cap`$:AB - ab$)), then the `W(AB)` edge is drawn to the matching node in `B`, as in the pure asynchronous case. Consequently, the modified version of `first-node(`α`,P)` constructed for pure asynchronous communication applies. Drawing the U-edge is equivalent to sending a message from `A` to AB (or to `B` itself, if the matching `W(AB)` is the topmost isolated node in `B`).

If AB is unable to contain the message (for message-capacity-limited buffers, when the buffer is full; in the case of byte-capacity-limited buffers, when size$(m) >$ (`cap`$:AB - ab$)), then `A` blocks on `W(AB)`, just as it does in the synchronous case when the topmost isolated node in `B` is not a `W(AB)`. This commstat is unblocked when `B` executes a `W(AB)` that releases a message slot, respectively a sufficient number of bytes, in AB. It should be noted that which synchronous cycles are partially asynchronous cycles depends on the sizes of the buffers, and the sizes of the messages sent by each commstat, and on these two sets of parameters alone. We shall abuse language slightly and simply refer to a partially asynchronous cycle, without mentioning buffer and message sizes, which are fixed for any one mfg construction.

When we indicate modifications to the operation of the algorithm in *Case 1b* below, we handle a blocked send as soon as the algorithm detects that enough space is free in the buffer to receive the message. This choice is not restrictive in any way, as we now note.

Lemma 7.3.1 *Partially asynchronous byte-capacity-limited systems of loop processes are uniform.*

Proof Sketch: Since we are dealing with loop processes, there is precisely one trace of each individual process. The finite buffer AB is uniquely associated with `W(AB)` actions. Executing receives may only increase the spare capacity of the buffer. Hence, when a blocked send is unblocked, it remains enabled until it executes, in any trace. An unblocked send that had been blocked in some trace cannot in that trace become part of a deadlock cycle, since none of the commevents of a deadlock cycle are ever enabled, by definition. We want to show that a blocked send commevent unblocked in some trace must also at some point in any trace be enabled, thus remain enabled until execution, thus never be part of a deadlock cycle.

The capacity of AB at any stage in a trace is a function $f(p, q)$, where p is the number of commevents of type `W(AB)` executed in `A`, and q is the number of such commevents executed in `B`. This function f is independent of which system trace is actually executed.

Suppose there were a commstat x of type `W(AB)` blocked and then unblocked in some trace T which is forever blocked in another trace T'. Suppose in T that the blocked execution of x is the p_0'th commevent of type `W(AB)`, at which time there have been q_0 commevents of type `W(AB)` in `B`. Suppose x is unblocked by the q_1'st execution of a commevent of type `W(AB)` in `B`. Then for $q_0 \leq i < q_1$, $(f(p_0, i) + \textsf{size}(x)) > \texttt{cap:}AB$, and $(f(p_0, q_1) + \textsf{size}(x)) \leq \texttt{cap:}AB$. Thus execution of T' never gets as far as the q_1'st such commevent, otherwise x would be unblocked in T' at that point. We claim (without proof) that the first such never-enabled commevent in `B` in T' is part of a partially asynchronous cycle in T'. (We may talk of cycles also in traces, by virtue of our technical definition of traces in Chapter 6.) Thus some commevent between the q_0'th and q_1'st commevents of type `W(AB)` in `B` is part of a deadlock cycle in T', but is not part of that deadlock cycle in T. This cycle is not an asynchronous cycle, else it would be a deadlock cycle also in T. Thus, it includes a blocked **send** y. However, y must eventually be enabled in T, since later commevents are executed in T. Further, $y \prec_{Ab} x$ (we omit the modification to the definition of $<_{Ab}$ necessary to obtain $\prec_{Ab}$, which among other things is a relation on commevents, not commacts). We started with a blocked-then-unblocked-in-T x which is never enabled in T', and concluded there was a $\prec_{Ab}$-earlier such commstat y. A standard argument leads to a contradiction with the fact that $\prec_{Ab}$ is a well-founded relation. Hence there can be no such commstat x. The lemma follows. ∎

Corollary 7.3.2 *If there is a deadlock cycle in a trace of a partially asynchronous system of loop processes with byte-capacity-limited buffers, then this deadlock cycle also appears in any trace in which an unblocked* **send** *executes immediately.*

We have thus justified our assumption in *Case 1b* below that an unblocked blocked-**send** may be handled immediately without inhibiting the mfg construction from finding all partially asynchronous deadlock cycles for given buffer and message sizes.

To illustrate how the algorithm works for partial asynchrony, we consider several different cases. We need to define not only isolated nodes as before, but also *pending* nodes. One may think of this intuitively in terms of two different colors of U edge, and in terms of the mfg algorithm as constructing a possible execution sequence of the system, as shown in Chapter 6. There are *black* U edges and *green* U edges. A black U edge intuitively represents a communication whose ⟨send, receive⟩ pair have both already been executed, and a green U edge represents an edge whose send has been executed, but whose receive is pending (i.e. the message is in the buffer). We may now define a *pending* node as a receive node with a green U edge drawn to it. Pending nodes remain isolated nodes, according to the definition of 'isolated', because they still need to be 'visited' by the algorithm to release the buffer capacity that the unread message was using up, to allow sends to unblock, or later sends not to block. Notice that this modification does not have to be considered for pure asynchrony, since sends never block in that case. Isolated nodes are those send nodes which have no U edge (of any color) drawn from them, together with receive nodes that either have no U edge or have a green U edge drawn to them. We now consider the modified behavior of the partial asynchrony algorithm. The subroutine `handle` must change, as must `ptin`. These changes are indicated informally.

Case 0: The topmost isolated node in `A` is `W(AB)`, and the matching commstat in `B` is the topmost isolated node of `B`. Draw a black U edge between the two nodes. Note we assume that the buffer capacity is large enough to contain the largest message that is transmitted. (If not, the system is bound to deadlock at the point at which this message is transmitted, and the analysis here is unnecessary).

Case 1: send nodes: The topmost isolated node y in `A` is `W(AB)`, but the matching commstat x in `B` is not the topmost isolated node of `B`.

- **Case 1a:** size$(y) \leq$ (`cap`:$AB - ab$). Draw a directed green U edge from y to x. (The next node lower than y in `A` thus becomes the topmost isolated node in `A`, and x is now pending.) After drawing the U edge, $ab \leftarrow ab +$ size(m).
- **Case 1b:** size$(y) >$ (`cap`:$AB - ab$). Then `A` 'blocks' (i.e. `handle` is called and no U-edge is drawn) and y remains the topmost isolated node in `A` until the first time that enough isolated `W(AB)`s in `B` are visited (note that 'isolated' includes 'pending' nodes) such that that $ab \geq$ size(m) (see Figure 7.4 and the case below). At this point the U edge is drawn from y: if x is topmost isolated in `B`

at this point, this U edge is black and ab is untouched; else the U edge is green and $ab \leftarrow ab + \mathsf{size}(m)$. After this operation, `handle` returns.

Case 2: receive nodes: The topmost isolated node x in `A` is a `W(BA)`.

Case 2a: x is pending. Let $\texttt{match}(x) = y$. Set $ab \leftarrow ab - \mathsf{size}(y)$, and change the color of the U edge to black.

Case 2b: x is not pending. Then the algorithm proceeds as in the synchronous case.

In the modifications above, the modifications to ab are a crucial part of the algorithm, involving the size of messages, which are expressed in terms of a size parameter $\mathsf{size}(y)$ whose argument is a commstat y.

Message-Capacity Limited Buffers. Suppose the capacity of a buffer is determined by the number of *messages* it can contain, rather than by the number of *bytes*. Systems in which messages tend to be the same size can be so modelled. In this case, $\mathsf{size}(y) = 1$ for all commstats y, and the increment and decrement operations using the size of the message are simple increment- and decrement-by-one.

Byte-Capacity Limited Buffers. For these, we have assumed that message size is determined by which send commstat is executed. This assumption, that if both m and m' are transmitted by commstat x, $\mathsf{size}(m) = \mathsf{size}(m')$, is required for uniformity. Figure 7.6 gives an example of code which may deadlock, or may not, if the message size is allowed to vary on repeated execution of the same commstat. The left-hand diagram shows a deadlocking trace. The buffer for W(AB) has size 2, and is unable to accept the second message from `A`, since this message has size 2 and the buffer only has one slot left, thus the system deadlocks at this point. By contrast, the right-hand example executes all commacts specified. Thus, with message size variation per commact, the communication is not uniform.

Relaxing the 'Static' Restriction. The constraint that message sizes are determined by which send commstat is executed may be considered a form of typing of messages. This typing may also allow analysis, although not an exact analysis, of general variable-length message passing in which the size of a message may vary arbitrarily, and is no longer bound to the commstat. Let's consider this typing in terms of the simplest case in which message sizes are construed simply in terms of *long* and *short* messages, and each commstat specifies whether a long or a short message is passed. Suppose firstly that an individual message has exactly one of these sizes, i.e. the message is packed into a byte string of exactly the right size, with padding as needed. This is the case we have considered, and deadlock analysis using the modified algorithm is exact.

Secondly, suppose that no packing is performed, that messages may vary according to the commact, but that the *long* and *short* designations are used as upper bounds to the length of the message that may be transmitted by the commstat. This case is also analyzable using the algorithm, but not exactly. The worst case is that each message is as large as the size limit for its type. The assumption is conservative, and using the algorithm we will always detect deadlock if it is present. However, if messages may be any size up to the maximum, the algorithm may determine that a system deadlocks, when it in fact does not, because the size of the messages is smaller than the size assumed. So, the algorithm may be used for a deadlock analysis even in the general case of byte-capacity limited buffers, although the analysis is conservative and may detect a potential deadlock even though an actual deadlock in a run is not guaranteed.

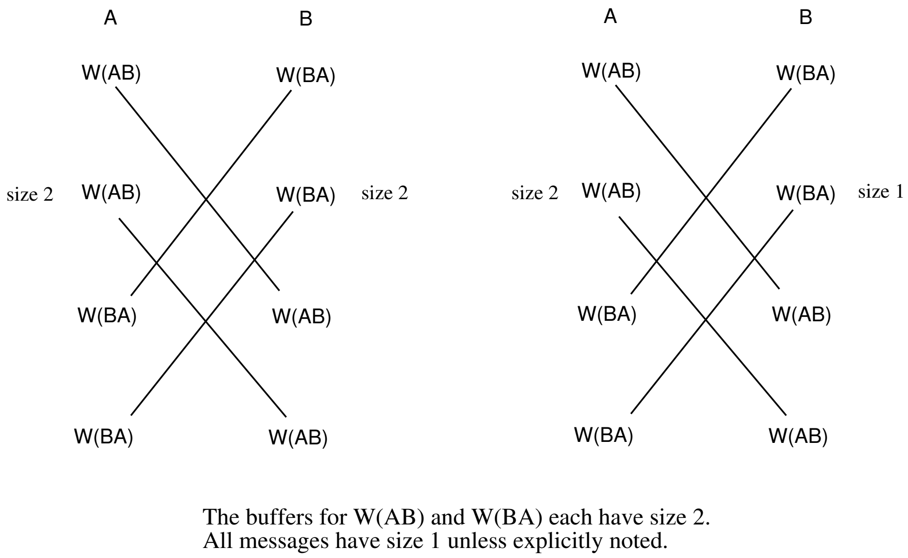


Figure 7.6: Deadlocking and non-deadlocking partially asynchronous communications.

Chapter 8

Multiway Synchronous Primitives

8.1 Generalizing Message Flow Graphs

For two process primitives, the mfg satisfies the condition (the *two-process unique-commstat condition*): every node is connected by a U edge to precisely one other node. Since the mfg is obtained from an expansion of the code defining each process, the meaning of the unique commstat condition is that when this code is executed, every commstat in the code for any process synchronizes with *a unique* commstat in another process, no matter when or how often it is executed. The synchronization can thus be illustrated by drawing an edge between the two commstats, as in our two-process message flow graph construction. We now consider what happens when we have primitives for k-process synchronization with $k \geq 2$ (which we refer to as multiway synchronization). We initially assume that all k processes must participate in a commact denoted by a k-process primitive. Later, in Section 8.4, we consider the consequences of relaxing this assumption.

8.1.1 Representing Multiway Synchronization

Say we have a communication primitive stating that five processes synchronize. An executable occurrence of this primitive will involve five commstats in these different processes. If we were to connect these nodes in the same way that we connected two-process nodes, we would need to use a *hyperedge* (a connection between, in this case, five nodes, instead of the usual two for a regular edge). Since hyperedges are difficult to illustrate, we indicate a multiway synchronization in code satisfying the unique-commstat condition by representing the hyperedge as a connected collection of U edges as follows. A distinguished node (called the *mother* node) is chosen and placed at the center of a star of ordinary U edges, which connect the mother to all other participating nodes. We call this an *edge-star*. The choice of mother node is technical and has no semantic significance for multiway synchronization. However, the edge-star representation is also useful to represent broadcast, especially asynchronous broadcast (i.e. non-blocking `send`, with separate atomic actions for `send` and each `receive`), in which the mother node may then be chosen to indicate the `send`. See Section 8.4.

We call such a construction a *generalized message flow graph* (g-mfg) if it satisfies the trace condition and the other conditions for an mfg, as well as the *(multiway) unique-commstat condition*: every node representing a k-process primitive is connected by an edge or an edge-star to exactly $k-1$ other nodes in other processes. Mfgs are also g-mfgs.

8.1.2 Some Vocabulary and Notation

We write an asymmetric three-process multiway synchronization between `A`, `B` and `C`, in which say `A` takes the role of sender, and `B`, `C` as receivers, as `W(A:BC)`, and similarly for k-process primitives. There are k different such primitives for a k-process synchronization amongst a given set of k processes. For deadlock analysis with synchonous communication, the asymmetry is unimportant - it is only important that the k primitives are distinguished from each other. The notation `W(ABC)` could also be used when there is just one primitive between the three processes.

We generalize the notion of matched pair to that of matched set as follows. The set of commstats $\{x, y, z, ...\}$ is a *matched set* if `label(`x`) = label(`y`) = label(`z`) = ...=` α, and all nodes with label α above x in `P(`x`)` and all nodes with label α above y in `P(`y`)`, and z in `P(`z`)`, ..., are already part of U edges. The g-mfg algorithm will ultimately place U edges in a star amongst elements of a matched set if no deadlock is found.

Assume that all finite loops are compiler deterministic and that they have all been unrolled. An *isolated node* is still a node that is not (yet) part of a U edge. A node is now said to be *matched* if it is part either of a U edge or of a matched set. As before, an isolated node becomes incorporated into a U edge (in an edge-star) as the algorithm proceeds, unless it cannot be matched with similar node in all of the other processes which partake in communication of that primitive type.

8.2 Deadlock Conditions in G-MFGs

The Cycle Condition remains unchanged as a condition for deadlock in the g-mfg. However, the Isolated Node Condition must be generalized to the *Incomplete Node Set Condition*: each node has a matching node in each of the other processes with which it participates in a commact. That is for every `W(A:BFG)` in, say, `A` there is either some statement `W(A:BFG)` in `B` that is not already assigned to a different commact, or such a node will appear as the loop body is duplicated; and likewise in `F` and `G`. If such a matching node is not found in one of `A`, `B`, `F` and `G`, the Incomplete Node Set Condition is not satisfied. The g-mfg algorithm returns an error message if the condition is found not to be satisfied, along with the g-mfg structure built up to this point. Jointly, the Incomplete Node Set Condition and the Cycle Condition provide necessary and sufficient conditions for there to be no deadlock.

8.3 The G-MFG Algorithm

The g-mfg algorithm is a modification of the mfg algorithm. We replace the service subroutine `incomplete-node-exit` with `isolated-node-set-exit`. The core routines `ptin` and `match`, which are modified to manipulate edge-stars instead of single U edges, process a set of matched nodes rather than a single matched node. The core routine `handle` remains the same, since U edges are still being drawn, albeit sometimes in edge-stars. The routines `ptin`, `match` and `incomplete-node-set-exit` are shown in Figure 8.1.

8.3.1 Special Techniques for Multiway Synchronization

A multiway synchronization hyperedge is represented in the generalized message flow graph as a star of U edges in which all nodes participating in a $\mathcal{V}$ communication are joined to a single node called $mother(\mathcal{V})$. The *mother* is selected normally by being the first node visited by the algorithm. Construing the multiway synchronization as a star-hyperedge enables us to use the same `handle` subroutine as used in the two-process case.

If $x, y \in hyperedge(\mathcal{V})$ and $x, y \neq mother(\mathcal{V})$, then there is an undirected path $\langle x, mother(\mathcal{V}), y\rangle$ between x and y consisting of precisely two

U edges, $\langle x, mother(\mathcal{V})\rangle$ and $\langle mother(\mathcal{V}), y\rangle$. Hence, all potential cycles involving hyperedges appear as cycles involving normal U edges, but unlike the two-process case in which a cycle may contain only a single U edge in between sequences of D edges, a cycle in the multiway case may contain such a two-edge subpath of U edges.

In the algorithm pseudocode in Figure 8.1, subroutine calls are written in normal (roman) script to distinguish them from functions, which are written in italic script.

8.3.2 How the G-MFG Algorithm Executes

Every event type is associated with a set of processes that participate in an event of that type. For example, the type `W(A:BDGH)` is associated with the set of processes {`A`, `B`, `D`, `G`, `H`}. The set of all processes participating in a communication of type α is denoted $processes(\alpha)$. We abuse notation also, for $x \in processes(\alpha)$, to let $processes(x) = processes(\alpha)$. (The algorithm is built upon the uniqueness of *processes(α)*).

A topmost isolated node x is selected, and its matches are found in the processes participating in that type of communication. The node x becomes the *mother* of the potential edge-star. The edge-star represents the hyperedge joining the processes in $processes(\alpha)$, and joins all the nodes in $(processes(x) - \{P(x)\})$ to x. If any of these matches is itself a topmost isolated node, say y, the U edge is drawn. If not, then the matched pair $\langle x, y\rangle$, representing part of the hyperedge, is put on the stack, and the topmost isolated node above y is processed by a recursive call. This recursive process can be thought of as in the mfg algorithm: a U edge is drawn only if that edge represents a communication action that *could* go next in a valid trace when only those two processes are considered. If not (i.e. there is an isolated node in $P(y)$ above y) then processing of the $\langle x, y\rangle$ pair is suspended, and processing of a node denoting an action that *must* be earlier than the action denoted by y (namely, the *topmost* such node) is started.

Note that a given hyperedge may be partly processed (by drawing U edges to nodes which happen to be topmost isolated nodes in their processes) before processing is suspended due to the discovery of some node which is not a topmost isolated node to which a U edge must be drawn from x.

In addition to popping the stack if a U edge can be drawn, the recursive calls may terminate with an error-routine call either due to discovering that there is no node of that type in one of the processes in $(processes(x) - \{P(x)\})$, or due to the discovery of a cycle.

```
ptin(u)::= { if P(u) ∈ Path then cycle-exit(P(u));
             Path ← Path ⌢⟨P(u)⟩;
             x ← tin(u);
             𝒱 ← match(x);
             mother(𝒱) ← x;
             { S ← 𝒱 − {x};
               while S ≠ ∅ do { select v ∈ S;
                                S ← S − {v};
                                if (preamble = true and v ∉ preamble(P(v)))
                                  then unroll(v);
                                handle(x, v)   }   };
             Path ← Path / ⟨P(u)⟩ }

match(x) ::= { α ← event-type(x);
               T ← processes(α) − {P(x)};
               match-nodes = ∅;
               while T ≠ ∅ do {
                     select P ∈ T;
                     if (there is no topmost isolated node of type α in P)
                       then if (preamble = true)
                              then incomplete-node-set-exit(x)
                              else if (there is a node of type α
                                            in loopbody(P))
                                     then duplicate(P)
                                     else incomplete-node-set-exit(x);
                     z ← first-node(α,P);
                     match-nodes ← match-nodes ∪ {z}   }
               return(match-nodes) }

incomplete-node-set-exit(x) ::= { print (“incomplete node set found:”, G, x);
                                  halt }
```

Figure 8.1: The Modified Routines For Building the G-MFG

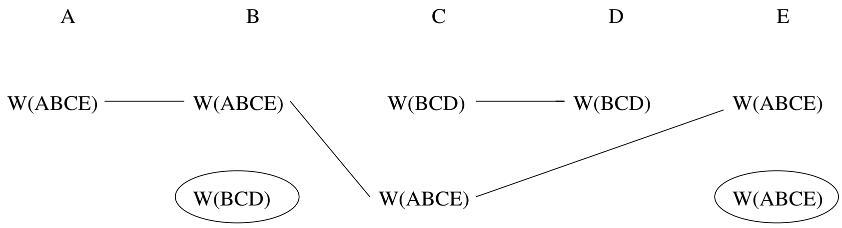


Figure 8.2: A Non-Uniform LOTOS-like Example: A deadlocking trace

8.4 Other Multiway Primitives

8.4.1 LOTOS synchronization primitives

The multiway synchronization primitives in LOTOS [ISO88] allow any *subset* of *processes*(α) to synchronize if they are able to, i.e. if Processes `A`, `B` and `C` all use gate g (the synchronization primitive), then `A` and `B` alone, `A` and `C` alone, `B` and `C` alone, or all three of `A`, `B`, and `C` may synchronize on g. However, if processes are able to synchronize then they must. This communication is non-transparent, and this non-transparency yields non-uniformity. The LOTOS synchronization primitives as defined in the standard may have a complex structure. When primitives are unstructured, the sublanguage defined is referred to as Basic LOTOS. It is sufficient for our purposes to use Basic LOTOS, and since it is a sublanguage of full LOTOS, the example carries over to the full language.

Lemma 8.4.1 *Basic LOTOS processes constructed using only sequential composition are non-uniform.*

Proof: In Figures 8.2 and 8.3 we give a deadlocking, respectively a non-deadlocking, trace of a simple system of LOTOS processes constructed using only the operation of sequential composition. The processes use two synchronization primitives, `W(ABCE)` and `W(BCD)`. The synchonizations in each trace are shown by U edges. In Figure 8.2, processes `C` and `D` synchronize first, followed by `A`, `B`, `C` and `E`, leaving the encircled isolated nodes in `B` and `E`. In Figure 8.3, processes `A`, `B` and `E` synchronize first, followed by `B`, `C` and `D`, followed by `C` and `E`. ∎

Lemma 8.4.2 *If a system* `S` *of communicating processes is non-uniform, there is no single mfg or g-mfg which describes all the traces.*

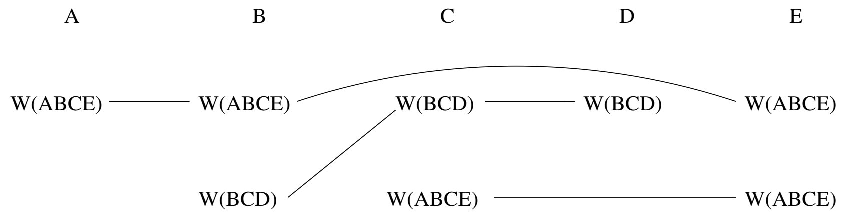


Figure 8.3: A Non-Uniform LOTOS-like Example: A non-deadlocking trace

Proof: It follows from Chapter 6 that all traces described by a single mfg or g-mfg deadlock, or all do not deadlock. Since `S` has a trace which deadlocks, and a trace which does not deadlock, no single mfg or g-mfg describes the set of all traces of `S`. ∎

Therefore a restriction to the Basic LOTOS synchronization mechanism is necessary in order to guarantee uniformity of a system of LOTOS loop processes. The g-mfg algorithm shows that a sufficient restriction is that to each primitive corresponds a fixed set of processes such that the execution of the primitive on every occasion gives a commact involving precisely all the members of this set.

Non-uniformity notwithstanding, all g-mfgs corresponding to any system of LOTOS processes may still be generated using a modified version of the g-mfg algorithm. With this modification, we claim the following without proof: different possible choices of topmost isolated node in `main` will generate different g-mfgs, but, conversely, all different possible g-mfgs are generated by some sequence of selections of topmost isolated node in `main`. We now indicate how to modify the g-mfg algorithm to analyze LOTOS processes subject to this indeterminacy.

In the modified version of the g-mfg algorithm, we no longer terminate with `incomplete-node-set-exit` or `cycle-exit` if an incomplete node set or a cycle is found. Firstly, observe that the mother x selected in `ptin` is a topmost isolated node, therefore is able to participate, and so must participate, in a synchronization on the primitive *event-type*(x). Other nodes in processses *processes*$(x) - \{P(x)\}$ may be unable to synchronize at this point. Hence, an edge-star with x as mother may nevertheless be drawn, omitting nodes in processes which are unable to synchronize. Recall that a U edge is drawn only when the two participating nodes are topmost isolated in their respective processes, i.e. they can synchronize at this point. Therefore processes which are unable to synchronize are discovered because a U edge may not be drawn. This happens in precisely two cases: (a) there is no node of type *event-type(x)* in that process; or (b) a cycle is discovered. The g-mfg algorithm in Figure 8.1 exits in either case, so in order to handle LOTOS primitives, we must modify the routines `ptin`, `match` and `handle` to continue under these conditions, in the following way.

If a cycle is detected by `ptin(`u`)`, then the current node is dropped from the current set `match-nodes` and the sets $\mathcal{V}$ and S, $P(u)$ is dropped from T, and `handle` returns without drawing a U edge. If there is no isolated node of the current *event-type* discovered in $P(u)$ by `match`, then $P(u)$ is removed from T and processing continues.

However, if it turns out that *every* possible choice of synchronization with x is either absent or leads to a cycle, then the g-mfg algorithm must exit drawing attention to the error. Hence we must add a guard so that if T ever becomes simply equal to $\{P(x)\}$, then `incomplete-node-set-exit` is called.

In order to claim that this or any series of modifications suffice to analyze code involving LOTOS primitives in the same manner as for two-process synchronization, it would be necessary to show that LOTOS loop processes are uniform. This would follow from the claim that at any point, there is a unique subset of processes $processes(x) - \{P(x)\}$ and a unique set of nodes in these processes (namely, the *first-node*s of type *event-type(x)*) that synchronize with x. We note that even if this condition were not to pertain, this case may still be handled (at some cost in processing time) by the backtracking technique below.

8.4.2 Backtracking to Handle Non-Transparency

In order to analyze a multiway nontransparent primitive, one could consider each primitive to be indeterminate amongst the possible transparent primitives obtained by considering all possible subsets of $processes(\alpha)$. In trying to handle such a primitive, then, all these subsets must be considered as possibilities. If in considering a possible subset, the algorithm discovers an incomplete-node-set, it does not exit, but merely eliminates the current subset from the list of possibilities for synchronization, and continues. There are obvious optimizations to use in picking the next choice of subset, for example, moving on to the subset of the current subset obtained by omitting the process from the current set of processes, as in the LOTOS case above. Similarly, if deadlock is detected, processing does not halt, but instead eliminates the current node and its process from consideration, and proceeds.

However, because of the *large* number of subset choices required, and the fact that all non-current processing alternatives must be retained on a stack in case backtrack should be necessary, we suggest that such a procedure would quickly run out of stack space and be for all intents and purposes impractical. We do not currently have alternative suggestions for handling non-transparent communication both deterministically and effectively.

Furthermore, if a backtracking method were employed, the mfg or g-mfg obtained could no longer be guaranteed to fulfil the condition investigated in Chapter 6, that the g-mfg is a finite representation of all valid traces. Instead, the g-mfg is a finite representation of some, not necessarily all, of the valid traces.

8.4.3 Asynchronous Broadcast Primitives

One may extend the algorithm presented here to asynchronous broadcast communication much more easily than to nontransparent communication. Asynchronous broadcast is handled in a manner similar to the extension to asynchronous communication in Chapter 7. In this case, rather than being selected in `ptin`, $mother(\mathcal{V})$ is in fact a function determined in advance: $mother(\mathcal{V})$ is the broadcaster, and the edges directed radially outwards from $mother(\mathcal{V})$. For example, $mother(\texttt{W(B:ACGH)}) = \texttt{B}$, and the U edges are $\langle \texttt{B},\texttt{A}\rangle$, $\langle \texttt{B},\texttt{C}\rangle$, $\langle \texttt{B},\texttt{G}\rangle$, and $\langle \texttt{B},\texttt{H}\rangle$. The `main` routine is modified to pass from a topmost isolated node selected for processing directly to the *mother* node in this communication before calling `ptin` on this *mother* node. `ptin` is modified simply to omit the statement assigning x to $mother(\mathcal{V})$.

With broadcast primitives, the mother now has semantic significance as the sender. Hence, an asynchronous cycle cannot include two consecutive directed edges from the same edge star, since these edges are all directed outwards from the mother. An asynchronous cycle in the multiway case contains alternating D and directed-U edges, as in the 2-process case.

Chapter 9

NP-Complete Problems

In this chapter, we show that determining if there exist a deadlocking system trace is NP-complete for an arbitrary number of communicating processes containing only six conditionals or six finite loops. Similarly, we show that reachability and determining the existence of non-deadlocking traces is NP-complete for an arbitrary number of similar processes, where 'six' is replaced by 'three'. We also show that reachability and determining the existence of non-deadlocking traces is NP-complete for only two communicating processes, if one consists of only a single chain (thread) and the other consists of a parallel DOALL of chains (threads). All the results hold for communications which are synchronous, asynchronous, or finitely-buffered with buffers of any size.

The *reachability problem* asks whether there is any trace, any run of the system, which will enter a specified global state. If there is such a trace, we say that the global state is *reachable*.

Reif and Smolka considered the reachability problem for systems of processes which communicate *static*ally (i.e. transparently, in our terminology) and asynchronously. They showed that the reachability problem for such systems is reducible to coverability of Petri nets (and vice versa), which is known to be decidable in exponential space, but requires 'nearly exponential' space unboundedly often [RS88]. The same paper contains a reduction of *3-SAT* to a communication problem that may seem similar to ours. However, they use an unrealistic model of message reception (M_2 in their notation) that never blocks on a read if a message has ever been sent on the corresponding channel. Furthermore, the reduction relies on multiple channels between a pair of processes, and uses explicit non-determinism.

We show here that reachability and deadlock problems are already NP-complete for very simple languages. Our proofs are simple, direct, and valid for synchronous, asynchronous, or finitely-buffered communications, using at most 2 channels per pair of processes. Reachability is NP-complete for systems in which each process has only communication statements, 3 conditionals, and no other code. A similar result holds for finite loops instead of conditionals. We show also for both cases that determining if there is a deadlocking trace is NP-complete (here we need only 6 conditionals). Reachability is also NP-complete for 2-process systems in which one process contains one simple parallel DOALL and the other has only communication statements.

If all processes have a halt command as their final statement, then the *final state* of the system is the state in which all processes have executed the final halt. Thus, the final state is reachable if and only if there is some trace in which the processes do not deadlock. We use this observation without further comment in the proofs. We ignore computation statements in the processes. The Boolean variables used in the NP-completeness proofs may be assumed to be input variables.

There are precisely two ways in which synchronously or asynchronously communicating systems can deadlock: either (a) one process reaches a commstat that has no partner (an *isolated node*); or (b) there is a *cycle*. An *asynchronous cycle* is a sequence $\langle x_0, ..., x_{k-1}\rangle$ of commstats such that either $x_{(j+1) mod\, k}$ is a successor of x_j if they are in the same process, or $x_{(j+1) mod\, k}$ is the target of a commact of which x_j is the source if they are in different processes. This implies that adjacent members of a cycle alternate between pairs in the same process and communicating pairs in different processes, since commacts involve a pair of commstats. A cycle represents an unsatisfiable temporal ordering of commacts. A *synchronous cycle* is defined the same as an asynchronous cycle except that a given pair of commstats defining one commact may appear in either order in the cycle (the pair forms an 'unordered edge' in the cycle). In an asynchronous cycle a blocked read commstat must always be a predecessor of a write that is in the same process and also in the cycle. In a synchronous cycle, either may occur first since all are blocked. An asynchronous cycle causes a deadlocking trace also if the communications are finite-buffered or synchronous. Conversely, if there is no synchronous cycle in any trace, then there are no asynchronous cycles either, and no cycles with finite-buffered communication. Our examples have either (a) traces that reach an isolated node, or (b) asynchronous cycles, or (c) neither of the above, and no synchronous cycles either. In cases (a) and (b), synchronous, asynchronous and finite-buffered communicating systems all deadlock, and in case (c) none of them do.

9.1 Traces that Deadlock

Communicating Processes with Conditionals. We consider m processes, each of which has exactly six commstats. The transformation is from *3-SAT* [GJ79]: Given a set of n variables (predicates), $p_1, p_2, \ldots, p_n$ and a set of m clauses, $C_1, C_2, \ldots, C_m$, with each clause containing precisely three literals, is there a truth assignment to the variables (predicates) such at least one literal evaluates to true in each clause?

Theorem 9.1.1 *Determining if there exists a trace that deadlocks is NP-complete for systems with an arbitrary number of communicating processes, each of which contains only six conditionals, commstats, and no other code. The result holds for synchronous, asynchronous, and finitely-buffered communication.*

Proof: To show that the problem is in NP, we guess the branches that cause deadlock. Each process now consists of only straight-line code, and it is easy to verify that there is an asynchronous cycle or a commstat without a partner.

To prove that the problem is NP-complete, given an instance of *3-SAT*, we construct a system of m communicating processes, M_1, ..., M_m. Process M_i corresponds to clause C_i, $1 \leq i \leq m$. See Figure 9.1 for a representation of the construction.

To simplify notation, we let $(i+1)$ denote 1 if $i = m$, and $(i-1)$ denote m if $i = 1$. Each process contains six conditionals. The first three correspond to the variables that occur in C_i; the second three correspond to the variables that occur in $C_{(i+1)}$. Suppose that p_j, p_k, and p_q are the three variables occurring in C_i. If the true (false) literal of p_j is used, then the *true* (*false*) branch of the conditional corresponding to p_j contains a single $\mathsf{W}(\mathsf{M}_{(i-1)},\mathsf{M}_i)$ followed by the next conditional. The *false* (*true*) branch goes directly to the same next conditional. Let p'_j, p'_k, p'_q be the three variables occurring in $C_{(i+1)}$. If p_q corresponds to the third conditional of C_i, then both branches terminate in a p'_j node. If the true (false) literal of p'_j is used, then the *true* (*false*) branch of the conditional corresponding to p'_j contains a single $\mathsf{W}(\mathsf{M}_i,\ \mathsf{M}_{(i+1)})$ followed by the next conditional. The *false* (*true*) branch goes directly to the same next conditional. If p'_q corresponds to the sixth conditional of C_i, then both branches terminate in a **halt** node.

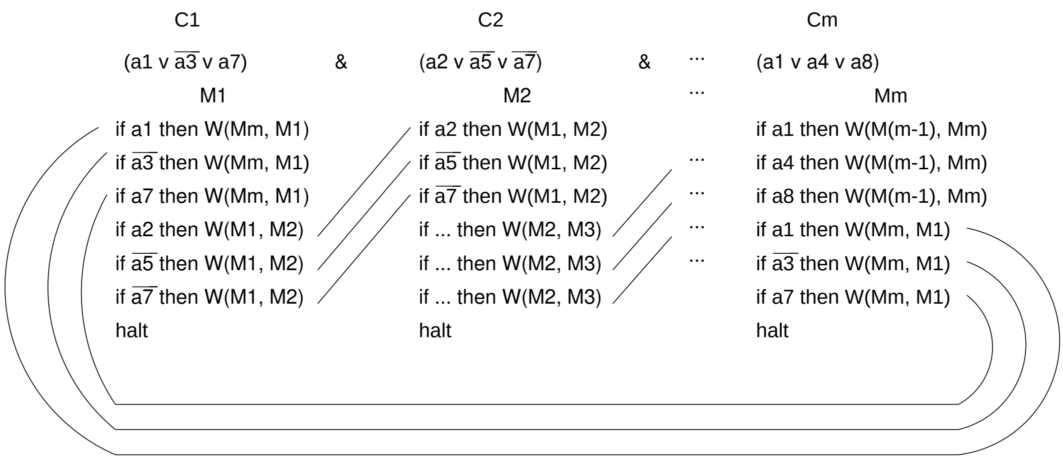


Figure 9.1: Encoding 3-SAT as a communication problem

If at least one of the variables in C_i and one in $C_{(i+1)}$ evaluates to *true*, then M_i will read first from M_{i-1} and then write to M_{i+1}. Consequently, if each clause has at least one variable that evaluates to *true*, i.e. there is a satisfying truth assignment for the instance of *3-SAT*, an asynchronous cycle involving all the M_is may be read off from the *3-SAT* instance, and a deadlocking trace may easily be written down from the cycle.

Conversely, since each process communicates precisely with its neighbors *mod m*, if there is an asynchronous cycle, it also must also involve all of the processes. Therefore, by observing which commstats are executed, we can derive a satisfying truth assignment for the instance of *3-SAT*. ∎

Communicating Processes with Loops. We say that a finite loop is *indeterminate* if the loop iteration count is not known at compile time. A straightforward modification of theorem 9.1.1 can be used to show that analyzing arbitrary numbers of processes each with six loops that execute either once or never is also NP-complete, giving:

Theorem 9.1.2 *Determining if there exists a trace that deadlocks is NP-complete for systems with an arbitrary number of communicating processes, each of which contains only six finite indeterminate loops each of which executes either* 0 *or* 1 *time, commstats, and no other code. The result holds for synchronous, asynchronous, and finitely-buffered communication.*

9.2 Traces that Don't Deadlock and Reachability

Communicating Processes with Conditionals. The transformation is from *One-in-Three 3-SAT* [GJ79]: Given a set of n variables (predicates), $p_1, p_2, \ldots, p_n$, and a set of m clauses, $C_1, C_2, \ldots, C_m$, with each clause containing precisely three literals, is there a truth assignment to the variables (predicates) such precisely one literal evaluates to true in each clause?

Theorem 9.2.1 *Determining if there exists a trace that does not deadlock, and reachability, is NP-complete for systems with an arbitrary number of communicating processes that contain at most* 3 *conditional branches, commstats, and no other code. The result holds for synchronous, asynchronous, and finitely-buffered communication.*

Proof: To show that the problem is in NP, we guess the branches that are taken and then verify that there is no isolated node or synchronous cycle.

To prove that the problem is NP-complete, given an instance of *One-in-Three 3-SAT*, we construct a corresponding system of $m+1$ communicating processes, M_0, M_1, ..., M_m. Clause process M_i corresponds to clause C_i, $1 \leq i \leq m$; master process M_0 is the only process with which the clause processes communicate. See Figure 9.2 for a representation of the construction.

Each clause process contains three branching nodes, one for each variable that occurs in C_i. Suppose that p_j, p_k, and p_q are the three variables occurring in C_i. If the true (false) literal of p_j is used, then the *true* (*false*) branch of the branching node corresponding to p_j contains a single $\mathsf{W}(\mathsf{M}_i,\mathsf{M}_0)$ followed by the next branch statement. The *false* (*true*) branch goes directly to the same next branch statement. If p_q corresponds to the third branch statement of C_i, then instead of having both branches terminate in another branch statement, both branches terminate in a **halt** statement. M_0 contains only straight-line code that consists of precisely m commstats of the form $\mathsf{W}(\mathsf{M}_1,\mathsf{M}_0)$, $\mathsf{W}(\mathsf{M}_2,\mathsf{M}_0)$, ..., $\mathsf{W}(\mathsf{M}_m,\mathsf{M}_0)$.

Suppose there is a satisfying truth assignment for an instance of *One-in-Three 3-SAT*. Then given this truth assignment, a trace may be read off in which each M_i communicates exactly once with M_0.

Conversely, we first note that any assignment in which none of the variables in clause C_k evaluates to *true* corresponds to a trace in which M_0 will wait forever to execute $\mathsf{W}(\mathsf{M}_k,\mathsf{M}_0)$; one in which more than one variable in C_k evaluates to *true* corresponds to a trace in which M_k will wait forever to execute the second $\mathsf{W}(\mathsf{M}_k,\mathsf{M}_0)$. In either case, there will be isolated nodes.

C1		C2		
(a1 v $\overline{\text{a3}}$ v a7)	&	(a2 v $\overline{\text{a5}}$ v $\overline{\text{a7}}$)	& ...	
M1		M2	...	M0
if a1 then W(M1, M0)		if a2 then W(M2, M0)		W(M1, M0)
if $\overline{\text{a3}}$ then W(M1, M0)		if $\overline{\text{a5}}$ then W(M2, M0)	...	W(M2, M0)
if a7 then W(M1, M0)		if $\overline{\text{a7}}$ then W(M2, M0)	...	...
halt		halt		...
				halt

Figure 9.2: Encoding One-in-Three 3-SAT as a communication problem

So any non-deadlocking trace corresponds to a satisfying assignment for the instance of *One-in-Three 3-SAT*. Given such a trace, the assignment may easily be read off the values of the Boolean variables in the trace. ∎

Communicating Processes with Loops. A straightforward modification of theorem 9.2.1 can be used to show that analyzing arbitrary numbers of processes each with three loops that execute either once or never is also NP-hard, giving:

Theorem 9.2.2 *Determining if there exists a trace that does not deadlock, and reachability, is NP-hard for systems with an arbitrary number of communicating processes that contain at most* 3 *finite indeterminate loops each of which executes* 0 *or* 1 *times, commstats, and no other code. The result holds for synchronous, asynchronous, and finitely-buffered communication.*

9.3 Processes with Parallel DOALL

We have shown that conventional branching constructs lead immediately to NP-completeness of reachability and deadlock properties. Branching may also be introduced in parallel code by a parallel DOALL, in which many control threads execute simultaneously. We show this also leads directly to NP-complete reachability and deadlock properties.

A *chain* is a sequence of commstats. The parallel DOALL construct applied to a set of chains creates a block (the body of the parallel DOALL), which consists of that set of chains, to be executed in parallel. The nodes in different chains are incomparable, but within a chain they are totally ordered.

The transformation is from the *3-partition problem* [GJ79]: given a multiset A containing $3n$ integers and a positive integer bound B, where $B/4 < a_i < B/2$ for all $a_i \in A$ and $\sum_{i=1}^{3n} a_i = Bn$, is there a partition of A into n triples of three elements each such that the sum of the integers in each triple equals B?

Theorem 9.3.1 *Let the system S_n consist of two communicating processes, one of which consists of only a single chain (straightline code) and the other of which consists of a set of $3n$ chains. Determining if there exists a trace of S_n that does not deadlock, and reachability, is NP-complete. The result holds for synchronous, asynchronous, and finitely-buffered communications.*

Proof: To show that the problem is in NP, we guess the pairs of commstats that match and then verify (using the linear time algorithm presented in [LS91] or [LS92a]) that there is no synchronous cycle.

To prove that the problem is NP-complete[1], we construct a system of 2 communicating processes, M_0 which consists of a single chain containing $2nB$ commstats, and M_1 which consists of $3n$ chains, one for each input integer. A commstat in M_0 can be matched with any of the available commstats in M_1. M_0 consists of B commstats of the form $\mathsf{W}(\mathsf{M}_0, \mathsf{M}_1)$ followed by B commstats of the form $\mathsf{W}(\mathsf{M}_1, \mathsf{M}_0)$. The pattern is then repeated $n-1$ times, giving a total of $2nB$ commstats. Chain C_i in M_1 consists of a_i commstats of the form $\mathsf{W}(\mathsf{M}_0, \mathsf{M}_1)$ followed by a_i commstats of the form $\mathsf{W}(\mathsf{M}_1, \mathsf{M}_0)$.

Given a solution to the *3-partition problem*, suppose that a_i, a_j, and a_k comprise one of the triples in the solution. We match the first B copies of $\mathsf{W}(\mathsf{M}_0, \mathsf{M}_1)$ in M_0 with the initial a_i, a_j, and a_k copies of $\mathsf{W}(\mathsf{M}_0, \mathsf{M}_1)$ in C_i, C_j, and C_k respectively. We then repeat the matching for the succeeding copies of $\mathsf{W}(\mathsf{M}_1, \mathsf{M}_0)$ in M_0 and C_i, C_j, and C_k. The matching is then repeated for each triple in the solution, and we have constructed a non-deadlocking trace.

Now suppose we are given a non-deadlocking trace. The blocks of size B in M_0 partition the chains. By the constraints on the values of the a_i, there are at least 3 chains that communicate with a given block in M_0. Let C_i, C_j,

[1] We assume the reader is familiar with the notions of *strong* and *weak* NP-completeness. Because the *3-partition problem* is strongly NP-complete, a reduction which is polynomial in the *value* of the numbers in the *3-partition problem* instance is sufficient for a proof of NP-completeness.

and C_k be chains communicating with the first M_0 block, and a_i, a_j, a_k the corresponding values. If $a_i + a_j + a_k > B$, there will be some unmatched $\mathsf{W}(\mathsf{M}_0,\ \mathsf{M}_1)$ comstats in at least one of the chains, say a_i. However, since $a_j + a_k < B$, the $\mathsf{W}(\mathsf{M}_1,\ \mathsf{M}_0)$ commstats in C_j and C_k do not match all B of the $\mathsf{W}(\mathsf{M}_1,\ \mathsf{M}_0)$ commstats in the next block of M_0. Therefore, the first unmatched commstat in M_0 and all non-empty chains of M_1 will be a **read**, resulting in asynchronous cycles. A similar argument holds if $a_i+a_j+a_k < B$. Thus, $a_i + a_j + a_k = B$, and the first two blocks of M_0 communicate exactly with the entire chains C_i, C_j, C_k. A trivial induction argument shows this happens for all succeeding blocks. By observing which 3 chains communicate with each block, we can construct a 3-partition. ∎ .

Chapter 10

Complexity

We have not yet noted the complexity of the message flow graph algorithm. In this short chapter, we consider some of the complexity questions.

First, the bad news. Define the *communication graph* to be the graph in which each process is represented by a node and there is an edge between a pair of nodes if there is some communication between them. Then, as is illustrated by Figure 10.1, a minimum message flow graph that is exponential in the size of the loop process input can be obtained even in the case in which the communication graph is a tree. However, in that case there is a polynomial time algorithm for deadlock detection that does not explicitly construct the minimum message flow graph [Sim]. In addition, the Numerical Matching Condition, which can be analyzed in time quadratic in the number of processes, provides a necessary (but not sufficient) test for deadlock detection. That is, if the Numerical Matching Condition is not satisfied, we know that the system will deadlock. In any case, it would appear prudent to write code in which exponential explosion in the size of the message flow graph does not occur. Given the difficulties in understanding and debugging message passing code, we conjecture that most code does not have a corresponding exponential size minimum message flow graph.

Now, the good news. Even though the mmfg in the worst case may be very large in comparison with the input, it is obtained in the easiest possible way by using our algorithm, as established by the following lemmas.

Lemma 10.0.2 *If there is no deadlock, then the running time of the algorithm is linear in the size of the message flow graph that is created.*

Proof: `handle` may be called a number of times on an individual isolated

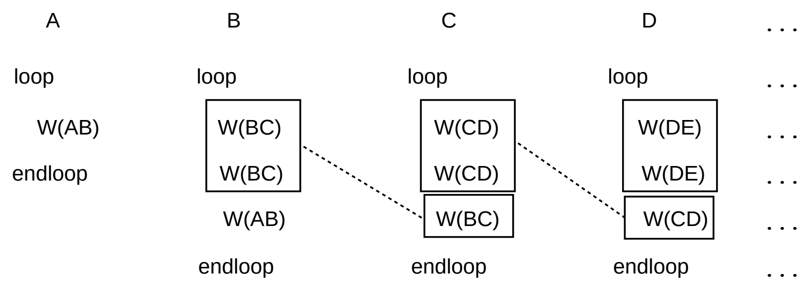


Figure 10.1: Input for which the minimum size message flow graph is exponential

node, but at the completion of each call, some topmost isolated node has been made part of a U edge. The total number of calls to `handle` is less than or equal to the total number of U edges created. Determining if $\mathrm{P}(x) \in$ `Path` can be implemented with $O(1)$ cost per call; therefore the total amount of work done between each call to `handle`, excluding the cost of `unroll`, is $O(1)$. `unroll` adds D edges between unrolled nodes in the loop code, but the number added is equal to the number of loop nodes unrolled, each one of which will become part of a U edge if the algorithm terminates without deadlock detection.

It is trivial to show that other subroutines are linear in the number of nodes in the final message flow graph. ∎

The following corollary contrasts with the NP-completeness results, which also assume that there are no infinite loops and any finite loops are iterated at most once.

Corollary 10.0.3 *If there is no infinite loop and if all finite loops are repeated only a constant, respectively polynomial, number of times, then the running time of the algorithm is linear, respectively polynomial, in the size of the problem input.*

Proof: After the loops have been unrolled, the number of nodes is linear, respectively polynomial, in the size of the problem input. The corollary follows from lemma 10.0.2. ∎

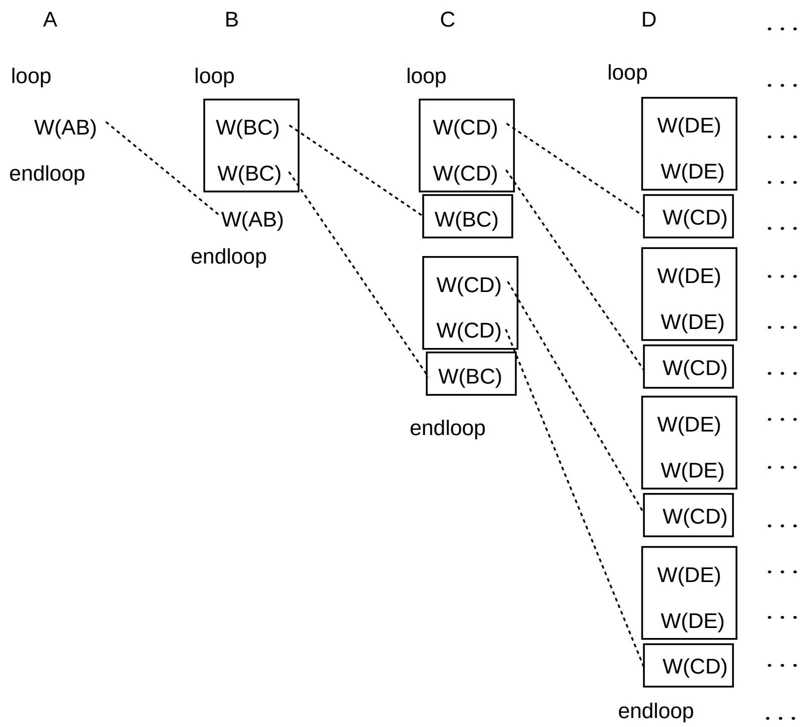


Figure 10.2: The exponential duplication

Chapter 11

Conclusions

We have presented an algorithm for statically analysing the structure of communications amongst concurrent processes. The algorithm produces a structure, the minimum message flow graph, which contains all the information concerning which statements in source code communicate with which other statements, and when. This information complements other static analysis methods such as data- and control-flow analysis.

Code appropriate for direct analysis using our algorithm is necessarily expressively limited, since some properties such as deadlock-freeness normally determined by a static analysis are PSPACE-complete or harder in the general case. In particular, our algorithm deterministically analyzes code in which there is a single trace determined by each input process. Each input process may in fact be regarded as a finite representation of a single process trace, so our method may be seen as the analysis of a collection of single-process traces.

We have proved the correctness of the algorithm, that it terminates and produces the desired output when it does so. This proof required a combinatorial condition, the Numerical Matching Condition, that is by itself easy to check (linear in the number of input statements and quadratic in the number of input processes) and may be used as a quick heuristic for determining whether deadlock will occur between communicating loop bodies.

We have extended the basic algorithm to handle asynchronous and partially asynchronous communication primitives, and multiway primitives such as broadcast.

We have also shown that the message flow graph constructed by our algorithm is minimal in a certain well-defined sense, and have shown that a sequentialization of the source code may be obtained directly from the message flow graph.

Our methods apply to the analysis of parallel code, or the analysis of software specifications written in the language of Structured Analysis, or to the analysis of proposed telecommunications protocols and services, without needing to develop a specific language in which to phrase our methods. Our algorithm may be implemented directly from the pseudocode in any procedural language with finite set and sequence data types; the operations needed on source code to produce input for the algorithm are simple. Our methods were developed with practical considerations in mind, and we hope that practioners of static analysis of concurrent and parallel systems will find them useful.

Bibliography

[Apt83] K. R. Apt. A static analysis of CSP programs. In E. Clarke and D. Kozen, editors, *Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 1–17. Springer Verlag, 1983.

[AS83] G.R. Andrews and F.B. Schneider. Concepts and notation for concurrent programming. *Computing Surveys*, 15(1):3–43, Mar 1983.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers*. Addison-Wesley, 1986.

[BCF$^+$94] Z. Boskus, A. Choudhari, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, Apr 1994.

[BCM90] J.R. Burch, E.M. Clarke, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of 5th Annual IEEE Syposium on Logic in Computer Science*, pages 428–439. IEEE Press, 1990.

[BFH91] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In E.M. Clarke and R.P. Kurshan, editors, *Computer Aided Verification: Proceedings of CAV'90*, volume 531 of *Lecture Notes in Computer Science*, pages 197–203. Springer Verlag, 1991.

[BFH$^+$92] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–269, Jun 1992.

[BP93] G. v. Bochmann and D. K. Probst, editors. *CAV'92: Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[Bry92] R.E. Bryant. Symbolic Boolean manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept 1992.

[Bv91] M. Barbeau and G. von Bochmann. Extension of the Karp and Miller procedure to LOTOS specifications. In E.M. Clarke and R.P. Kurshan, editors, *Computer Aided Verification: Proceedings of CAV'90*, volume 531 of *Lecture Notes in Computer Science*, pages 333–342. Springer Verlag, 1991.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, Apr 1983.

[CD92] D. Cohen and N. Dorn. An experiment in analyzing switch recovery procedures. In M. Diaz, editor, *FORTE92 conference on Formal Description Techniques.* Participants' Proceedings, 1992.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Apr 1986.

[CK91] E.M. Clarke and R.P. Kurshan, editors. *Computer Aided Verification: Proceedings of CAV'90*, volume 531 of *Lecture Notes in Computer Science.* Springer Verlag, 1991.

[CS87] J.E. Cuny and L. Snyder. Conversion from data-driven to synchronous execution in loop programs. *ACM Transactions on Programming Languages and Systems*, 9(4):599–617, Oct 1987.

[Die90a] V. Diekert. *Combinatorics on Traces*, volume 454 of *Lecture Notes in Computer Science.* Springer Verlag, 1990.

[Die90b] V. Diekert, ed. Proceedings of the Workshop of the EBRA-Working-Group 3166 on Algebraic and Syntactic Methods in Computer Science. Technical Report TUM-I9002, Technische Universität München Institut für Informatik, 1990.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W. H. Freeman, 1979.

[GW92] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In K.G. Larsen and A. Skou, editors, *Computer Aided Verification: Proceedings of CAV'91*, volume 575 of *Lecture Notes in Computer Science*, pages 332–341. Springer Verlag, 1992.

[GY92] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, Dec 1992.

[Haf92] J. Hafner. Personal communication, 1992.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An overview of the Fortran D programming system. Technical Report COMP TR91-154, Rice University, Houston, TX, Mar 1991.

[HKT91] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report COMP TR91-149, Rice University, Houston, TX, 1991.

[HKT94] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, Apr 1994.

[HL95] D. Hogrefe and S. Leue, editors. *Formal Description Techniques, VII.* Proceedings of the Seventh International Conference on Formal Description Techniques. Chapman & Hall, 1995. To appear.

[HM91] D.P. Helmbold and C.E. McDowell. Computing reachable states of parallel programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 76–84. ACM Press, 1991.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[Hol72] R.C. Holt. Some deadlock properties in computer systems. *Computing Surveys*, 4(3):179–196, Sep 1972.

[Hol88] G.J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice and Experience*, 18(2):137–161, Feb 1988.

[Hol91] G. J. Holzman. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[HP88] D.J. Hatley and I.A. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House, 1988.

[ISO88] ISO. Information Processing Systems - Open Systems Interconnection - LOTOS : A formal description technique based on temporal ordering of observed behavior, International Standard 8807. International Standards Organisation, 1988.

[ISO91] ISO. OSI Conformance Testing methodology and framework, International Standard 9464. International Standards Organisation, 1991.

[Jac86] M.A. Jackson. *System Development.* Prentice Hall International, 1986.

[JK91] R. Janicki and M. Koutny. Using optimal simulations to reduce reachability graphs. In E.M. Clarke and R.P. Kurshan, editors, *Computer Aided Verification: Proceedings of CAV'90*, volume 531 of *Lecture Notes in Computer Science*, pages 166–175. Springer Verlag, 1991.

[Jon87] G. Jones. *Programming in Occam.* Prentice Hall International, 1987.

[KM69] R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[Lam90] L. Lamport. Correspondence on the concurrency mailing list, 1990.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LL94a] P. B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 1994. To appear.

[LL94b] P. B. Ladkin and S. Leue. What do Message Sequence Charts mean? In *[TAU94]*, pages 301–316. 1994.

[LL95] P. B. Ladkin and S. Leue. Four issues concerning the semantics of Message Flow Graphs. In *[HL95]*. 1995. To appear.

[LS91] P.B. Ladkin and B.B. Simons. Compile-time analysis of communicating processes. Technical Report RJ 8488, IBM Almaden Research Center, Nov 1991.

[LS92a] P.B. Ladkin and B.B. Simons. Compile-time analysis of communicating processes. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, pages 248–259. ACM Press, 1992.

[LS92b] K.G. Larsen and A. Skou, editors. *Computer Aided Verification: Proceedings of CAV'91*, volume 575 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[LS94a] P.B. Ladkin and B.B. Simons. Static analysis of multiway synchronization. In *Proceedings of CASCON '94: Integrated Solutions*, Toronto, Canada, October-November 1994. IBM Software Solutions Toronto Laboratory and the National Research Council of Canada. To appear.

[LS94b] P.B. Ladkin and B.B. Simons. Static deadlock analysis for CSP-type communications. In D. Fussell, editor, *Responsive Computer Systems: Toward Integration of Fault-Tolerance and Real Time*. Kluwer, 1994. To appear.

[Maz87] A Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri-Nets, Applications and Relationship to other Models of Concurrency, LNCS Vol. 255*, volume LNCS 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer Verlag, 1987.

[Mer90] N. Mercouroff. An algorithm for analyzing communicating processes. Technical Report LIX/RR/90/12, Ecole Polytechnique, Laboratoire d'Informatique, Palaiseaux Cedex, France, 1990.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[Nar91] P. Narendran. Personal communication, 1991.

[PP91] W. Peng and S. Purushothaman. A unified approach to the deadlock detection problem in networks of communicating finite state machines. In E.M. Clarke and R.P. Kurshan, editors, *Computer Aided Verification: Proceedings of CAV'90*, volume 531 of *Lecture Notes in Computer Science*, pages 243–252. Springer Verlag, 1991.

[PP92] W. Peng and S. Purushothaman. Analysis of a class of communicating finite state machines. *Acta Informatica*, 29:499–522, 1992.

[RS88] J. Reif and S.A. Smolka. The complexity of reachability in distributed communicating processes. *Acta Informatica*, 25:333–354, 1988.

[RS90] J. Reif and S.A. Smolka. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1):1–31, Feb 1990.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. MIT Press, 1989.

[Sar92] V. Sarkar. A concurrent execution semantics for parallel program graphs and program dependence graphs. In *Conference Record of the Fifth Annual Workshop on Languages and Compilers for Parallel Computing*. Yale University, Department of Computer Science, August 1992. Technical Report YALEU/DCS/RR-915; also Springer Verlag, to appear.

[Sch90] F.B. Schneider. Personal communication, 1990.

[SCS85] F.B. Schneider, R. Conway, and D. Skeen. Thrifty execution of task pipelines. *Acta Informatica*, 22:35–45, 1985.

[SF93] B. Simons and J. Ferrante. An efficient algorithm for constructing a control flow graph for parallel code. Technical Report TR.03.465-1, IBM Santa Teresa Laboratory, San Jose, CA, 1993.

[SH89] N. Sridhar and C.A.H. Hoare. JSD expressed in CSP. In J. Cameron, editor, *JSP and JSD: The Jackson Approach to Software Development*, pages 334–363. IEEE Computer Society Press, 1989.

[Sim] B.B. Simons. A polynomial time algorithm for compile-time deadlock analysis when the communication graph is a tree. In Preparation.

[SPP91] A. Silberschatz, J. Peterson, and Galvin P. *Operating System Concepts, Third Edition.* Addison-Wesley, 1991.

[SS94] V. Sarkar and B. Simons. Parallel program graphs and their classification. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallelism*, volume 768 of *Lecture Notes in Computer Science*, pages 633–655. Springer Verlag, 1994.

[TAU94] R. L. Tenney, P. D. Amer, and M. Ü. Uyar, editors. *Formal Description Techniques, VI.* IFIP Transactions C, Proceedings of the Sixth International Conference on Formal Description Techniques. North-Holland, 1994.

[Tay83] R.N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[WM86] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems, Vols I-III.* Yourdon Press - Prentice Hall International, 1985-86.

[YG92] T. Yang and A. Gerasoulis. Pyrros: Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, pages 438–446. ACM Press, 1992.

[YT88] M. Young and R.N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, Oct 1988.

Index