Uppsala Master's Theses
in Computing Science 108
Examensarbete MN3
1997-04-15
ISSN 1100-1836

Towards a deadlock analysis for Erlang programs

Richard Carlsson

Computing Science Department
Uppsala University
Box 311
S-751 05 Uppsala
Sweden

Supervisor: Håkan Millroth
Examiner: Håkan Millroth
Passed:

Abstract

We argue that there is a need for automated verification of absence of deadlocks in concurrent programs. We briefly describe how absence of deadlocks can be showed, in general, for a program, how the necessary information can be gathered, and the use of formal methods. We discuss timeouts as a method of run-time deadlock detection, and why they remain necessary in some situations, and we suggest a strategy for modular verification of subsystems of concurrent programs.

We then give an overview of the area of process communication, nondeterminism, and synchronization problems, with focus on the semantics of ERLANG, and give a quick introduction to the ERLANG language. We give detailed models of the semantics of the ERLANG messaging system and its synchronization primitives both in terms of synchronously communicating sequential processes and of Concurrent Constraint Programming.

Lastly, we review a deadlock analysis method which we have found likely to be able to fulfil our requrements, and discuss the particular problems with adapting it to ERLANG.

Contents

Chapter 1

Introduction

ERLANG is a functional programming language that supports lightweight concurrent processes without being dependent on the operating system it runs under. Concurrency is explicit in the language, and processes communicate via the single, uniform method of asynchronous message passing. This work addresses the problem of process communication deadlocks in programs written in ERLANG, or languages with similar properties.

1.1 Overview

In the following sections of this chapter, we describe the problems of deadlock and other communication anomalies in reactive systems, deadlock detection and recovery, and static verification of absence of deadlocks. We give requirements on a development tool for static checking of possible deadlocks, and suggest a strategy for modular verification of absence of deadlocks. Lastly, we discuss related work, present a summary of our results and suggest future work.

In chapter 2 we briefly describe how absence of deadlock can be shown in general, how to gather the necessary information, and how to identify deadlocked program states. We also discuss formal methods.

Chapter 3 discusses the asynchronous and synchronous variants of process communication, and the sources of nondeterminism in concurrent (and sequential) programs, such as process scheduling and data streams attached to more than one producer.

In chapter 4 we give an introduction to the ERLANG language, including some of its background.

In chapter 5 we show how the asynchronous inter-process communication system of ERLANG can be modelled using synchronous communication primitives, and Concurrent Constraint Programming, respectively, in order to see to what extent analyses developed for these latter concurrency models apply to ERLANG programs.

Chapter 6, lastly, discusses those analysis methods we consider as likely to be useful for deadlock analysis of ERLANG programs, and in particular describes the method developed by Masticola [19], our extension of that method [4], and the primary problems expected in implementing this method for ERLANG programs.

1.2 The deadlock problem

ERLANG is very suitable for constructing large reactive systems containing any number of processes, such as telephone exchanges or similar control systems, server applications, and simulations. A problem with these kinds of systems is that the communication pattern between processes can often be extremely complex and difficult to predict. Even in simple cases with a small number of processes, an error can be very hard for a human to discover without actually running the program – and once detected, it is not necessarily a simpler task to find its cause.

Deadlock and starvation

Apart from the type of errors where messages are sent to the wrong destinations (which are usually comparatively easy to discover by examining the program), there are, informally speaking and following the common terminology, two classes of synchronization anomalies where expected messages never arrive: *starvation*, where no potential sender process can exist, and *deadlock*, where one or more processes are all simultaneously waiting for a message from some other process involved in the same deadlock, forming at least one dependency cycle. Deadlocks, and in particular how they can be detected, are the main focus of interest of this work.

The types of programming errors causing starvation and deadlock, respectively, are (again, informally) of quite different character. Starvation occurs when a process is waiting to synchronise, and no other process is able to reach a program point which could complete the synchronization, for reasons such as processes terminating abnormally, loss of information (e. g. if no other process has access to the communication channel), or an error in the flow of control or data. Deadlock, on the other hand, is due to misjudgement of the communication pattern between processes, ending in an unexpected combination of individual process states, and is by far the more difficult to foresee or detect.

Suspension and deadlock

A process is said to be *suspended* while it is waiting to synchronise, and often a distinction is made between *suspension* (of a machine state), where no process in the state is able to proceed, and *local suspension*, where at least one process in the state may execute, but for some subset of processes, all members of that set are suspended and cannot synchronise with any other process in any reachable state. Typically, a message from an input device will allow the computation to continue. One example of local suspension is a terminal program waiting for input in a multitasking system where other processes are executing meanwhile.

Deadlock can be seen as a special case of suspension, where no input from external devices can enable any process to proceed. *Local deadlock*, more commonly known as *livelock*, is the corresponding case for a set of locally suspended processes. In real-world systems, the local form is largely the more common, and thus we want to be able to cover all such cases as well as any global deadlocks. In the following, the term "deadlock" will be used to denote local as well as global deadlock, except where stated otherwise.

Deadlock detection

There are two approaches to the problem of verifying that there are no occurrences of deadlock in a program: the usually more straightforward is *dynamic detection*, where the running program is examined at intervals by the system, looking for possible deadlocks. The other approach is *static detection*, where the program code is analysed beforehand in order to verify that no deadlocks can arise in any possible execution of the program.

Traditionally, deadlocks are a matter of operating systems studies, and are commonly described in terms of resource allocation: a deadlock occurs when each in a set of processes has allocated one or more resources, and is waiting for access to one or more further resources that are held by some other process or processes in the same set. Here, also, there is at least one circular dependency. In general, a cycle in the corresponding dependency graph is a necessary condition for a resource allocation deadlock, and as we have shown [4], this is also true for process communication deadlocks in the cases that are of interest to this work. Therefore, dependency cycles appear to be a better characteristic than any particular description in terms of communication or resource allocation.

Dynamic detection is the generally used method of handling deadlocks in operating systems, for several resons. First, static analysis is usually not even an option in this case, because of the *ad hoc* manner in which new executing program code is generally added to the system, and of the high frequency of such additions: each time program code has been added, a new analysis is required of the interaction between the added program and those programs it could communicate with or compete with for shared resources. Secondly, it is easy to track resource ownership and demands of processes, and there are fast standard algorithms for finding cycles in the dependency graph. Third, a resource allocation deadlock can often be remedied by forcing some processes to release their held resources, so that other processes may complete.

Recovering from deadlocks

In the case of process communication, however, dynamic detection is not much good for anything other than discovering the possible existence of a deadlock. The description in terms of resources is not suitable in this case, because signals or messages can be seen as parts of the computation performed by the processes. If process p suspends when trying to receive a message from process q, it is not because q currently "owns" the message (and could possibly be forced to release it), but because the message has not actually been computed yet (including it being sent), and nothing less than completing that computation could give another process access to it.

Thus, even if the program execution could be rolled back arbitrarily far, there is no general solution for avoiding the deadlock, once discovered. Another process scheduling might work, but finding such a scheduling – if one exists – cannot be done without some form of advanced program analysis. (Even though run-time information could be used which would not be available to a fully static analysis, it is hardly practical to apply program analysis once a deadlock has actually occurred.) It can also be argued that a program that depends on the process scheduling for behaving as intended is an incorrect program, or at least a flawed one.

The method of handling deadlocks generally adopted in application programming is to use timeouts – equipping process suspensions with an upper time limit, after which some fixed action is taken – to detect possibly deadlocked processes, and e. g. having a supervising system kill and restart them (and possibly others associated with them) under the conservative assumption that the expected data cannot be produced. The obvious drawbacks are the risk of losing information, the time spent shutting down, restarting and resynchronising processes, and the possibility of avalanche effects causing large parts of the system to be restarted. Also, this strategy is of no help in locating the actual error, and does not guarantee that the deadlock does not reoccur, even immediately.

A tool for static verification

We therefore assert that the only way of consistently handling process communication deadlocks, which is potentially powerful enough to avoid such drawbacks, is static program analysis and verification – if this can be done with sufficient precision to prove, for a significant subset of all correct programs, that there is no possibility of deadlock. As the general problem of checking whether a program will deadlock or not is, of course, undecidable, a safe approximation is the best that can be done, i. e., a program will be regarded as having the potential to deadlock unless the opposite can be proved.

In systems where *hot code loading*, i. e., the loading of new program modules, or new versions of previously loaded modules, is possible, the situation becomes similar to that in an operating system. However, software modules are seldom loaded which communicate with previously loaded ones in such a way that a deadlock might ensue. The modular verification strategy discussed further below should keep the frequency and extent of necessary deadlock analyses on a practical level, even in this case.

It is important that a deadlock analysis have very good precision in order for it to be a useful tool. If false cases of deadlocks are reported too often, it will confuse more than help the programmer, and possibly force her to rewrite the program in an overly explicit way, to allow the analysis to verify it. Preferably, the analysis should also, for the cases it cannot verify, give good indication of what parts of the program could possibly cause a deadlock, and under which conditions.

The aim of this work has been to seek out a static analysis method for ERLANG programs, which could realistically be developed into a useful programming tool as we have described; or at least specify the requirements on such a method.

1.3 Fail-safe systems and run-time deadlock recovery

In many real-world, real-time control systems, it is imperative that the system remains active no matter what errors may occur. Any subsystem that terminates abnormally, hangs, or deadlocks, must be restarted within the shortest time possible. It can be noted that the higher the level of abstraction, the more important it gets that each subsystem can be restarted with as little disturbance as possible in the other subsystems. For example, when recovering from an error in a hardware controller process, it might be allowable to restart several

associated processes, if it is the easiest way to recover, and any errors induced by this are negligible. In comparison, an error in a file server in an operating system should not cause other major parts of the system to be restarted, if avoidable.

As outlined above, the common programming practice for handling possible deadlocks in ERLANG applications programming is to include a timeout limit (see section 4.3) in attempts to receive a message. If the expected message does not arrive within this time limit (on the scale of ten seconds, or less in very time-critical systems), it is assumed that it never will. The timed-out process then terminates, usually informing some controlling instance of this, so it may be restarted if necessary for the functioning of the system. Effectively, the timeout is transformed into a proper error, causing the process to terminate abnormally. (More advanced schemes may be thought of, where resynchronization is first attempted, but they must ultimately rely on this mechanism.)

Why timeouts are necessary

All this is a relatively simple matter, but it is tedious and clutters the program. However, it appears that this mechanism must necessarily be included in all mission-critical programs. If an error is detected in some process (e. g. if it signals abnormal termination), it is in general not possible to decide which other processes were depending on it. If suitable, all possibly depending processes may be killed, and the whole subsystem restarted; otherwise, it has to be trusted that all depending processes will time out eventually (and relatively soon).

Reversely, given that a message is missing, it cannot in general be decided which process was supposed to supply the message, unless the correspondence between receivers and possible senders in the program is of a simple nature. In particular, if the error lies in the flow of control or data in such a way that the intended sender does not raise an exception in some way, then the timeout may be the only indication of the error that will ever appear during run-time. A process attempting to receive a message is effectively left on its own.

Apparently, unless the program is *proven* to be fully correct, we cannot be rid of these timeouts if we want our program to be fail-safe. Also, if messages may be transferred over a communication link, it can never be guaranteed that they will eventually reach their destinations, and so even software verification is not sufficient for removing the timeouts. It would however be desirable to automatise the adding of timeouts to the program, either as a source-level transformation or as a feature of the run-time system, e. g. by equipping all possibly suspending program statements with a default timeout limit and handler except where such have already been explicitly specified by the programmer. For important special cases, such as server processes, software packages already exist (e. g. the ERLANG `gen_server` module) which hide most of the error handling involved.

1.4 Program verification – why and when?

If, as we argued above, the timeouts and recovery mechanisms cannot be removed from mission-critical systems, for what reason would we want to go through the trouble of verifying that such a system is free – or partially free – from deadlocks?

As previously mentioned, recovering from deadlocks and other errors obviously causes a temporary drop in the system throughput. If deadlocks occur often, the impact on the general efficiency can be great. As the probability of synchronization errors is a lot higher than that of other errors, it can be expected that a system which is at least partially verified to be free from deadlocks will have better performance than a more *ad hoc*-constructed system, and naturally be less likely to lose information.

Also, deadlocks are different from other programming errors, in that they are in general significantly more difficult to reason about. The program code for the internal work of any process concerns "only" the incoming data and the corresponding production of output data. In contrast, the code handling the intercommunication of a number of processes is distributed over all those processes. To the programmer, this is the only part of the code where concurrency really makes a difference, because parts of a system may change state more or less independently. Traditional debugging tools are of little help in finding errors in the communication pattern, because they do not regard the interaction of processes, beyond straightforward data flow.

It can therefore be expected that of any undiscovered serious errors in a well tested system, the majority have to do with synchronization. When such errors appear in a running system, they are hard to locate, and when attempting to correct the program it can be very difficult to avoid introducing new errors. An efficient verification tool could lower the probability of deadlocks to the same as that of other software errors, or even less.

Non-critical applications

Nor is it the case that every real-world program is mission-critical. For most program tools, we are more or less satisfied to know that the likelihood of software errors is small; even non-concurrent programs are in practice never verified to be error-free. In contrast, speed usually remains important whatever the application, and the handling of timeouts is somewhat time-expensive – in particular when several may be pending simultaneously: both the maintaining and the handling of those which are triggered. For such non-critical programs, we want to avoid using timeouts in as many places as possible, without inordinately increasing the risk of unrecoverable errors occurring. For achieving this, static verification is necessary.

So, if we can ignore the possibility of lost messages, and are not particularly concerned with quick detection and recovery of any and all errors, where can timeouts be omitted? To see this, we list the different cases of sender anomalies:

- *Abnormal termination.*
- *Control or data flow errors.* (The sender does not terminate, but will never send the expected message.)
- *Starvation.* If the sender is starved (not deadlocked) it is not really an error in the sender itself, but in the processes upon which it is depending. Inductively, starvation is never the actual source of the error.
- *Deadlock.* (The sender is involved in a deadlock, that might or might not include the receiving process of this discussion.)

Of these, deadlock is then the only case where timeouts are a necessary precaution, since the remaining actual error sources do not pertain to the synchronization itself. Timeouts can thus be omitted for all possibly suspending program points which can never be involved in a deadlock; also, for all possible deadlocks (circular waits) it is sufficient to assure that at least one involved synchronization statement is always able to time out and break the deadlock.

1.5 A modular strategy for verification

Verifying absence of deadlocks for a large program will of course, like any difficult program analysis, take relatively long time to perform. If the program is very large – on the order of a hundred thousand lines of code, as in many important industrial applications – the time needed might well be forbiddingly long. Also, even if the verification would finish in reasonable time, it would be impractical to be forced to reanalyse the whole program each time a (possibly quite small) part of it has been modified.

What we would like to be able to do, is to build larger systems from smaller subsystems of intercommunicating processes that can be treated as independent modules[1] with well-defined interfaces, and verify for each individual module that no deadlock can occur in which all involved processes are part of that module only. In other words, if a deadlock occurs which includes processes that are part of the module, then at least one process in the deadlock lies outside the module.

Obviously, when a system is constructed from internally deadlock-free modules, we want to verify that the composed system is itself free from internal deadlocks, so that it can be treated as a new module. Such deadlocks can then only occur in the communication between the modules forming the composed system. This fact should allow analysis of a complex system to be simplified, by first recursively analysing its component modules in isolation, and then analysing only the interaction between these modules. In addition, when the program is modified, only those modules containing a changed component need to be reanalysed.

1.6 Related Work

Masticola gives in [19, Chapter 10] a thorough survey of the field of deadlock analysis and detection methods up until 1993, covering both formal and more practically oriented methods. We discuss Masticola's own work in depth in chapter 6.

Cheung and Kramer [5] give an unsafe analysis (i. e., one that underestimates the actual behaviour) for distributed systems, which is computationally cheap and could be used in a development tool, especially since the underestimation assures that the programmer is not bothered with false alarms. Particularly, the authors suggest, such a tool would be useful in the early, tentative stages of protocol specification.

Many data dependency analyses for concurrent logic programs, eġ. Debray *et al.* (1996) [11] assume a fixed scheduling of processes (goals), or are in other

[1] The term "module" used here in a general sense, and not specifically referring to the module system in ERLANG.

ways too restricted to be used for deadlock analysis of general ERLANG programs. Debray (1994) [12] gives a framework for data flow analysis of concurrent logic programs, but as it depends on the use of a substitution-closed domain, it cannot be used to reason about dependencies between variables (messages).

Codish, Falaschi and Marriott give in [6] (published 1994) a suspension analysis for concurrent logic programs, based on confluence of reduction orders, and in [7] (conference proceedings, 1993) together with Winsborough extend this to a confluent semantics for concurrent constraint programs. The method can be extended to a deadlock analysis, but is not applicable in practice to ERLANG programs because of the nondeterministic merging of message streams; see section 5.2.2.

More lately, Matthews [20] (1995) showed that absence of deadlock in data flow networks with lazy pipes can be proved without referring to an operational semantics – an "extensional" proof. Stoller and Schneider [23] (1995) give a Hoare-style proof system for programs using causally-ordered [18] message-passing.

Colby [9] gives an analysis of the communication topology of concurrent programs, using abstract interpretation [10]. The analysis is non-uniform, meaning that it distinguishes between iterations in infinite recursive patterns [13], and works by relating pairs of processes. The results can be very precise even for programs that have a recursive structure with dynamic creation of processes and channels, and precision is not necessarily reduced when channels are passed in messages. The method is not immediately applicable to the problem of deadlock analysis, because it does not relate processes in a wider context, but could play an important part in demonstrating and/or eliminating possible dependencies between processes.

In [8] (unpublished), Colby gives a framework for determining both synchronization and aliasing properties for a concurrent functional language, using Deutsch's [14] lattice of prefix relations on strings, where the strings are taken from regular languages for control paths and data. The method can be said to generalise that in [9], from the observation that aliasing and synchronization are mutually dependent, and can be used to show a large variety of difficult synchronization properties of programs, apparently with very good precision. Colby gives an example showing that advanced communication-topology analysis can be done in polynomial time, but no results from automated analysis of real programs have been reported as of this writing. It is unclear whether the method can realistically be used on large programs for the purpose of verifying absence of deadlocks.

Over the last decade, much work has been done in the field of distributed data bases, trading systems, operating systems, and similar, regarding the reliability of such systems. The concept of *process groups* (see e. g. [3]) has a lot in common with the modular verification of subsystems we suggested in section 1.5.

1.7 Summary of results

In this thesis, we have first argumented for the need of automated verification of absence of deadlocks in concurrent programs, discussed when and where time-outs remain a necessity even if such verification has taken place, and suggested a strategy for modular verification of subsystems of concurrent programs.

Generally, we have given a wide overview of the area of process communication and synchronization problems with focus on the semantics of ERLANG, and we have given models of the semantics of the ERLANG messaging system and its synchronization primitives both in terms of synchronously communicating sequential processes and of Concurrent Constraint Programming. This work has pointed out interesting particularities of the different concurrency and synchronization models, and showed upon the non-applicability to ERLANG programs of analyses designed for these specific concurrency models.

Lastly, we have reviewed Masticola's method [19] of verifying absence of deadlock by detecting and excluding possible deadlock cycles, discussing its applicability to ERLANG programs. As a side-effect of this work, we have extended Masticola's method so as to make its application to programs in ERLANG and similar languages theoretically sound, and giving it a more abstract shape [4].

1.8 Future work

Of most interest to us would be the implementation of our extension [4] of Masticola's method [19], for programs in ERLANG or a similar language, and the practical application of the modular approach to verification of reactive systems we suggested in section 1.5. It would then be very interesting to see if Colby's communication topology analysis [9] could be used to improve the precision of the deadlock analysis, and in particular, if the problem of dynamic process creation in the program to be verified can be efficiently and automatically handled using the information yielded by such a topology analysis.

Of particular interest would also be to see a practical implementation of Colby's synchronization analysis [8], to see if it is realistic for large, real-world programs.

1.9 Acknowledgements

Many thanks to: my parents for all their support, my tutor Håkan Millroth for being patient[2], and my sister and my friends – Marcus in particular – for putting up with my lugubriousness and keeping me going, basically.

No thanks to: the system.

[2] Hofstadter's Law: It will always take more time than you expect, even when Hofstadter's Law has been accounted for.

Chapter 2

Showing absence of deadlocks

Any conservative information regarding where a program might deadlock, can be viewed as a description of a set of machine states, containing at least all reachable states which may contain a (local) deadlock. If the described set can be shown to be empty, the program is free from deadlocks. Ideally, only reachable deadlocked states would be included by the description, but for programs in general the problem is not decidable.

2.1 Finding reachable program states

The "traditional" and straightforward way of finding the above set is to inductively generate it from the base set of possible initial states for the program. For any state already in the set, those states reachable from that particular state (given the program text and the language semantics), called the *successors* of the state, are added to the set, until no new states can be added. Then, any states which can be shown to definitely not contain any deadlocks may be removed. (See [24] for one of the earliest examples of this method applied to concurrent programs.)

For certain limited types of systems this approach, called *state enumeration*, is practical, but for concurrent programs in general it quickly becomes intractable. The set of reachable states for a program is often infinite, in which case the above procedure will not terminate. Also, experience has shown (see e. g. [19, Section 7.6]) that for large programs, even if the number of reachable states is finite, it is commonly much too great to be handled directly.

Instead of the actual program states, the enumeration is therefore in the general case performed over abstract descriptions of states, or rather, sets of states. If the abstraction is chosen suitably, the process will terminate in reasonably short time, at the cost of a loss in precision, i. e., more states will (in general) be described by the generated set of abstract states than can actually be reached by the program.

In connection with the above, we take the opportunity to mention a widely used, very general method of program analysis, called *abstract interpretation* [10], which we will not describe in more detail, but which can be used to gather any

kind of (finite) information regarding properties of the different points of execution of a program, such as the possible (abstract) machine states at each program point. Abstract interpretation can be seen as an extreme abstraction of state enumeration, tracing every execution of the program over abstract descriptions of its states.

2.2 Identifying deadlocked states

To identify globally deadlocked machine states is not difficult – they are precisely those which have no successors, but do not represent the termination of the program. In order to identify a locally deadlocked subset of processes, however, it must be shown that from a certain state on, all of the processes in the subset will remain suspended indefinitely. More to the point, to show for any particular suspended process that it is *not* locally deadlocked, it must be shown that the process *will necessarily* be enabled to continue its execution in some subsequent state (assuming process scheduling is fair), something which is very difficult to do in practice.

For useful application to real-world programs, a deadlock analysis must be able to verify absence of local deadlocks, but the apparent problem is that of finding an abstraction which guarantees finiteness without losing too much in precision. The wider the approximation, the more difficult it becomes to rule out the possibility of deadlock in the described states.

2.3 Formal methods

With a "formal method" is generally meant the proving of some specific property of a particular program, by means of mathematical-logical deduction, either by hand or automatically. The results are, of course, neither more nor less formally valid than those of e. g. a state enumeration method, and the distinction between formal and other methods is not defined. Often, though, it is implied that a formal method yields exact solutions in those cases where it terminates with a solution. Usually, automatised proof methods require the user to supply information such as loop invariants, which makes the application of analysis to a program a non-trivial task.

Formal proof methods for concurrent programs is a quite young field, even within computing science. Most existing methods are effectively built on state enumerations, verifying formally for each state that it cannot contain a deadlock, and thus have the same practical limitations as state enumeration methods, apart from generally being an order of magnitude more time-consuming than these.

See also section 1.6 for related work on formal methods.

Chapter 3

Process communication

The different strategies for process communication in concurrent languages can be divided into two classes: *asynchronous* communication and *synchronous* communication. Each can easily be described in terms of the other, and there is no general consensus as to which is the more basic of the two.

3.1 Asynchronous communication

When information (anything from an arbitrary message to a simple signal) is transferred asynchronously, the sender does not wait for the information to be accepted by the receiver, but immediately continues execution, which might include the sending of further messages. The receiver will independently attempt to accept a message when it needs the information, often entering a waiting state which could last for any length of time, until suitable data is delivered. In a completely general implementation of asynchronous communication, any number of messages may be sent regardless of whether previous messages have been accepted or not, the only limitation being the amount of space available in practice for message buffering.

The most studied form of asynchronous communication is that of Concurrent Logic Programming, or in more modern terminology, Concurrent Constraint Programming (CCP). For a short introduction to CCP, see section 5.2.1.

3.2 Synchronous communication

When two processes communicate synchronously, each independently enters a state where it is waiting for the other to become ready to exchange information. Usually, the data is transferred in a single direction, where one process is specifically attempting to send and the other to receive. Synchronous communication is used in several concurrent imperative languages such as Ada, Concurrent C and Concurrent ML, its main advantage being that no implicit buffering needs to be done by the run-time system. In fact, asynchronous communication can be seen as buffered synchronous (unidirected) communication, where the sender only suspends if there is no room in the receiver's message buffer. (In the completely general case, the buffer is assumed to be infinite).

On the other hand, synchronous communication can be seen as two-way asynchronous communication in two steps. First, the sender dispatches its message and enters a waiting state. When the receiver has accepted the message, it dispatches an acknowledge signal and proceeds with its own execution. Lastly, the sender receives the acknowledge signal, and can itself proceed.

The description of this procedure displays the largest drawback of synchronous communication, namely, that almost twice as much housekeeping needs to be done by the run-time system, which slows down execution. Furthermore, and more importantly in practice, if the communication takes place between processes which exist on different nodes in a network, twice as many messages need to be transferred over the comparatively very slow communication links, where in a majority of the cases a one-way, asynchronous transfer would have sufficed.

Also, synchronously communicating concurrent programs can be said to be twice as sensitive to deadlocks and starvation, compared to asynchronously communicating ones, because there are twice as many opportunities for processes to become indefinitely suspended, if either of the sending or the receiving processes should be malfunctioning, or messages be lost.[1] Generally speaking, synchronously communicating programs are much tighter coupled than asynchronously communicating ones, and this is an additional drawback when constructing large programs, since it makes them not only sensitive to errors, but also difficult to modify.

Furthermore, it is apparent that in synchronous communication, additional data must be associated with each sent message, in order to tell how to properly return the acknowledge signal.

3.3 Nondeterminism and process scheduling

There are two basic sources of nondeterminism in the semantics of concurrent programs: first, versions of the nondeterministic *choice* construct, and secondly, nondeterministic process scheduling (reduction order) in situations where processes exchange data.

Actual implementations of course always make deterministic choices, but they are free to choose any strategy that falls within the space of nondeterministic behaviour in the language specification. (Often, fairness is a requirement; see section 4.2.)

Nondeterministic choice

A choice construct consist of a fixed, finite, set of condition–action pairs, or *clauses*. The conditions are in practice tests on variable data, and the corresponding action may be executed/computed only if the condition evaluates to *true*. The evaluation of a condition should not change the program store/variable bindings. In the general case, conditions may overlap, i. e., more than one may evaluate to *true*, but only one action may be selected for evaluation. If all conditions evaluate to *false*, the evaluation of the construct itself fails.

[1] In some implementations, the system will either return the acknowledge signal to the sender, or cause it to fail, if it is the case that the message cannot be transferred, i. e., if effectively the communication channel is closed.

The following is the "canonical" example of how a nondeterministic choice construct can be used, expressed in the syntax of ERLANG:

```
max(P) ->
    case P of
        {X, Y} when X <= Y  ->  Y;
        {X, Y} when X >= Y  ->  X;
    end;
```

`max(P)` yields the maximum of a pair `P` of numbers. If the numbers are equal, any clause matches (and in this example, they yield the same result if that is the case). Note, however, that in ERLANG, the semantics of the `case` construct (and its variant, the `if`) is not actually nondeterministic – the clauses are tried in the textual order.

A nondeterministic choice could be used to select any message currently in the receive-buffer, which matches a specific pattern, thus making the order of accepted messages nondeterministic. In ERLANG, however, the `receive` construct matches the buffered messages in order of delivery; see section 4.3.

Nondeterministic process scheduling

When more than one process can affect the value of a data item, the result of the program execution may depend on the order in which processes are executed. The program semantics is then said to be *non-confluent*.

For instance, suppose that two processes `A` and `B` exist, such that both are ready to execute, where `A` will eventually reach a program point where it sets a global variable `X` to the value `1`, and similarly, `B` will eventually set `X` to `2`. Suppose ready processes may be selected for execution in any order. A third process `C`, also ready to execute, which will read `X` and take its action depending on the found value, can then receive any of three values: `0`, `1` or `2`, assuming that `X` initially had the value `0`. Basically the same situation occurs with *multiple-writer streams*, if more than one process is ready to send on a single channel, when some process is ready to receive on that channel.

In addition, whether processes are executed atomically from the point when they are selected by the scheduler, until they explicitly enter a suspended state or terminate, or their execution may be interleaved with that of others (which is of course the more general case), can further affect the outcome of the program execution.

In ERLANG, since there are no global variables, and all variables must be bound before they are referenced (i. e., unbound variables may not be passed as arguments), the message-passing system is the only place where process scheduling may affect the semantics of a program (not considering real-time effects). We can modify the above example so that process `C` is waiting for a message, while both `A` and `B` are ready to execute, and will eventually send messages `1` and `2`, respectively, to `C`. Which message is actually received by `C` depends on the scheduling, and on the implementation of the message-passing system.

However, it turns out that the scheduling of processes in ERLANG (assuming it meets the fairness requirement) is basically irrelevant to the semantics of ERLANG programs. The reason for this is that the message buffering and the possibility of transfer delays completely hides all and any effects of process scheduling (see chapter 5 for details). Thus, unless we study a model in which

we have placed additional restrictions on the behaviour of the message-passing system, we need not consider the effects of different process schedulings.

Chapter 4

Introduction to the Erlang language

ERLANG was originally invented as a programming notation to be automatically translated into a concurrent logic language. The first implementations used Parlog [16] as the target language; this was then changed to Strand [15] for efficiency reasons. (Section 5.2.2 discusses the translation from ERLANG to languages such as these.) Both of the latter languages inherit most of their syntax and terminology from Prolog [22] and CSP [17], and a lot of that has carried over also to ERLANG. Later implementations of ERLANG are however built on abstract machine models designed specifically for its own particular semantics.

Unlike Parlog and Strand, ERLANG is a functional language. It has constructs for explicit sequentialization of evaluations (as is common in practical functional languages), and it is strict, i. e., all parameters to a function call are evaluated before the call is performed. Variables are single-assignment only and must be bound before they are used in an expression. There is no destructive updating in the language.

It should be noted that as of this writing, ERLANG is still under much development, and that what is stated here applies primarily to the language as described in the 1996 edition of "Concurrent Programming in ERLANG" [2]. Still, for the purposes of this work, we are only concerned with the core of the language, which is not likely to change in any significant way.

4.1 The functional core

The basic syntax is similar to Prolog. Identifiers whose first character is a capital letter are automatically interpreted as variables, and each occurrence of a single '_' (underscore) character represents a distinct and anonymous variable. Atom names can be surrounded by single quotes in order to contain otherwise unallowed characters. A character sequence surrounded by double-quotes is shorthand notation for the corresponding list of character code integers. Numeric literals have the expected syntax.

ERLANG is dynamically typed. This implies that the type of the value assigned to a certain variable in some function is not necessarily the same over all possible invocations of that function. There are four types of primitive values:

- Atoms ("terms" in the terminology of mathematical logic)
- Numbers (integer or floating-point)
- Process identifiers (or "Pids")
- References (automatically generated system-unique objects)

In addition, there are two forms of compound data types: *tuples* {`X1, X2, ... Xn`} and *lists* `[X1, X2, ... Xn | Remainder]`, where it should be noted that `Remainder` need not be another list (although it usually is). The form `[X1, X2, ... Xn]` denotes the list `[X1, X2, ... Xn | []]`. Such lists, terminated by the empty list `[]`, are called *proper* or *well-formed*. In ERLANG terminology, a *term* is a data element, i. e., an element of a primitive or a compound data type.

There is no functional type. Function references can be passed around and used by giving the name of the function as an atom, together with its arity. In general, two functions with the same name are considered distinct if they have different arity, so it is no error to define, say, `f(X)` and `f(A, B)` in the same program.

Variable binding and function evaluation

Pattern matching is the basic way in which variables become bound. An assignment `X = Expression` is just a special case of the pattern matching primitive `Pattern = Expression`. Its result, assuming the match succeeds, is the value of `Expression`, allowing constructs like `X = {A, B} = E`, the '=' operator being right-associative.

Together with *guards*, pattern matching also provides the means for case selection in the language. A function is defined by a sequence of clauses, whose heads specify patterns to be matched with the passed arguments, optionally qualified by guards. Guards are sets of tests (arithmetic and elementwise comparisons and built-in test functions) on terms. Such terms may contain arithmetic operations and calls to a small set of built-in functions, but not e. g. calls to user-defined functions, since a guard test may not cause a side-effect in any way.

The arguments of a function call are matched sequentially against the clause heads that define the function, and if a match succeeds, the guard tests (if any) of that clause are evaluated; these normally involve variables occurring in the argument patterns, but may not introduce new variables. The evaluation order of a set of tests is not defined. If the guard succeeds (or is empty), the clause is selected and its body is evaluated. Otherwise the next clause in turn is attempted. There is no backtracking; if no clause matches, the program fails.

A clause body consists of an expression to be evaluated. Comma is used as a left-associative infix sequencing operator which evaluates its left argument before its right; its result is that of the right argument. In all other cases, the evaluation order of arguments is not defined.

An example

Figure 4.1 shows a simple example of a program in ERLANG (Armstrong *et al.* [2, program 1.1]). Only explicitly exported names can be referenced from outside the module; such a reference would be written `math1:factorial(N)`.

```
-module(math1).
-export([factorial/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N - 1).
```

Figure 4.1: ERLANG program example.

In their most primitive form, clause-matching languages require a new function (or predicate, as it be) to be written for each situation where different cases are handled. To simplify programming, ERLANG provides the `case` and `if` primitives, where `case` offers the full pattern matching with optional guards, and `if` is a shorter form with guards only. These can be viewed as anonymous functions with a solitary definition and use. It should be noted that there is no boolean data type, and that the use of the keyword `true` for catch-all (empty) guards in `if` constructs is merely a syntactical convention.

This is then an alternative way of defining the factorial function in figure 4.1:

```
factorial(N) ->
    if
        N == 0 -> 1;
        N >  0 -> N * factorial(N - 1)
    end.
```

(Actually, this version is stricter, since it uses the guard `N > 0` in the second case. To be equivalent to the first definition, this guard should be `true`.)

4.2 Concurrency

The built-in function `spawn` causes the evaluation of a function as a separate process, yielding the Pid for the new process. The originating process does not wait for its completion (but can be informed of this via signals, where desired). When the top-level function of a process has been evaluated, the process terminates. The result computed by the function is lost. Data are transferred between processes using the message mechanism described below.

The scheduling of processes in an ERLANG implementation is not fixed, but must fulfill two requirements. It must be *fair*, i. e., a process that is ready to run must eventually become scheduled, and, secondly, no process may run for more than a short period of time if there are other processes ready to run. This latter requirement exists for practical purposes only (*viz.*, for short response times in real-time systems), and holds no importance to our discussion. The period, known as a *time slice*, is typically in the range of a few milliseconds or less.

4.3 Inter-process communication

Communication in ERLANG is asynchronous. The primitive '`!`' (send) is an infix operator whose left argument is the Pid of the receiver process and whose right argument is the message to be sent. A message is any constant value (term). The result of the expression `Pid ! Message` is the value that was sent.

```
receive
    <pattern 1> [when <guard 1>] ->
        <actions 1>;
    ...
    <pattern N> [when <guard N>] ->
        <actions N>;
[after <timeout-expression> ->
    <timeout-actions>]
end
```

Figure 4.2: The syntax of the `receive` primitive.

The send operation proceeds immediately without even waiting for the message to be delivered to its destination (which could be on another computer). If the receiver process has terminated when the message is sent, this does not affect the sender in any way, and the message is simply lost. Otherwise, the message is stored in the mailbox of the receiver process.

The order in which messages are delivered is not necessarily the same as the order in which they were sent, taken over time. (In particular, this is noticeable when messages are transferred over a network, in distributed implementations.) The language only guarantees that messages with the same sender and receiver processes will be delivered in the same relative order as that in which they were sent (often referred to as first-in-first-out, or FIFO, ordering). Primarily, this implies that message passing cannot be interpreted as an atomic operation, and may be preempted before the message is delivered to the receiver's mailbox. Just as the process scheduling is required to be fair, it is also required of implementations that sent messages will eventually be delivered (or be lost, e. g. if the receiver has terminated).

Receiving messages

Delivered messages are kept in the mailbox in incoming order, and are not removed until explicitly received by the process. To receive a message, a set of pattern/expression pairs is specified, and the mailbox is searched in order from older to newer for messages that match one of the patterns. Each message is matched against all patterns in sequence before the next message is tried. If a message is found to match a pattern, it is removed from the mailbox and the corresponding expression is evaluated. If no message in the mailbox matches any of the patterns, the process will suspend until such a message is delivered or a timeout occurs.

All this is handled by the primitive `receive`. Its full syntax is shown in figure 4.2, where `<timeout-expression>` evaluates to an integer or the atom `infinity`. An integer value represents a time measured in milliseconds. A value of zero causes the timeout to occur immediately after the current contents of the mailbox have been searched, while `infinity` inhibits the timeout completely. Leaving out the `after` part is equivalent to specifying `infinity`, so the `receive` will then terminate only when a matching message is received. If no patterns are specified, no message can be matched, effectively turning the construct into a delay. Like the similar `case` primitive, the "actions" are the expressions which

will be evaluated depending on the selected case, and the value of the primitive itself is the same as that of the last expression evaluated within it.

Message priorities can be implemented by nesting one `receive` primitive within the `after` part of another with a zero timeout value, causing a complete search of the mailbox for a certain set of patterns before any other patterns are attempted or new messages waited for.

4.4 Other features

ERLANG includes several other features, not important in principle to our discussion. The most significant of these are the module system, which allows program modules to be loaded or updated dynamically, the global Pid registration database, process dictionaries, links and signals, error handling, run-time code replacement, and ports.

Chapter 5

Translational semantics of Erlang

Most work on analysing process synchronization behaviour (in fact, almost all) has been directed at synchronously communicating processes and at Concurrent Constraint Programming (Concurrent Logic Programming). In order to see where such results and methods can be applied to systems of ERLANG processes, we need to describe the semantics of ERLANG programs, and in particular the asynchronous inter-process communication, in terms of these concurrency models. In such a description, there is not necessarily a one-to-one correspondence between the ERLANG processes and processes in the target model.

5.1 Synchronously communicating processes

A central property of a system of (individually deterministic) processes communicating by synchronous message passing only, is that it restricts all nondeterminism to the choices of sender and receiver in synchronizations. It is thus not dependent on, e. g., the scheduling of processes in the same way as a system where processes test and assign values to shared variables. (Usually in message-based systems, processes do not in general share data, and messages are regarded as copied upon sending, as in ERLANG itself.) This property allows communication analyses to focus on the spawning (and termination) of processes, and on the synchronization events. At all times, every data item in the system is locally stored in some process, and common data flow analysis methods can be used for most of the part.

5.1.1 Synchronous communication primitives

For the purpose of this chapter, we will assume the existence of two primitives for synchronous communication: $send(c, M)$ and $receive(c)$, where c is a channel and M a message (any value). A process executing *send* or *receive* over a channel c will suspend until it can be synchronised with another process. If two processes exist in the same state such that one executes $send(c, M)$ and the other $receive(c)$ for the same c, both processes will be able to proceed, M being the result of the *receive* primitive. For our discussion, we can leave the result of the *send*

primitive undefined at all times. If the channel is closed, we assume that both primitives proceed immediately, the result of a *receive* undefined.

Note that we do not make any assumptions about the uniqueness of sending and receiving processes here; it is possible that for a specific channel, several processes may be waiting simultaneously to send or receive. The semantics of these primitives are nondeterministic in this respect, and any send/receive pair may be chosen for reduction. In a model of an ERLANG system, however, there will be at most one receiving process for any channel.

5.1.2 Modeling the semantics of the message passing

For communication analyses to be conservative (safe), the model being used must correspond to the most general interpretation possible of the language semantics. From the description in section 4.3 of message passing in ERLANG, we extract the following central facts:

- An ERLANG process must never be blocked unless it explicitly executes a `receive`. In particular, execution of the asynchronous send ('!') primitive must always be able to complete, regardless of whether any earlier sent messages have been delivered or not.
- The transferring of a message from its sender to the mailbox of the receiver cannot be assumed to be an atomic operation; all program execution may proceed for arbitrarily long while a particular message is in transfer, with the exception that any subsequent messages having the same sender and destination as that message will not be delivered before it (so-called FIFO ordering).

From the first point, it is apparent that a message, once sent, must be described as being carried by a process separate from the sender and receiver processes, in a synchronous model. To see this, suppose the receiver process is involved in an arbitrarily long (or nonterminating) computation containing no `receive` call. It cannot then for any reason execute a *receive*, since this would cause it to suspend indefinitely if no sender existed. (We assume that there is no possibility of "polling".) Thus, any attempt to transfer a message to that process (executing a *send*) would block until the process reached a `receive`, or possibly forever. To allow the asynchronous send to complete immediately, the message must be handed over to a carrier process, which can synchronise with the receiver at a later time.

From the second point above, it can be deduced that there must be at least one carrier process for each pair of source and destination processes. Consider the program in Figure 5.1. (The result of the call `self`() is the Pid of the executing process. Also, recall that the result from a `spawn`() call is the Pid of the new process.) The result from evaluating the call `nondet:a()` may be either of `t1` or `t2`, since there is a possibility (in particular if the processes are running on distinct nodes in a distributed implementation) that `t1` is delayed during transfer, so that meanwhile, `t2` is transferred to `B` via `C`. Eventually, `B` will then receive `t1` and pass `t2` back to `A`.

This example shows that there must be a separate carrier process for each source–destination pair, or otherwise `t2` could not be transferred to `C`, nor forwarded to `B`, before `t1` has been delivered.

```
-module(nondet).
-export([a/0]).

a() ->
    A = self(),
    B = spawn(nondet, b, [A]),
    C = spawn(nondet, c, [B]),
    B ! t1,                     % Send one message to B
    C ! t2,                     % ... and one to C
    receive M -> M end.

b(Pid) ->
    receive M1 -> M1 end,       % Receive two messages
    receive M2 -> M2 end,
    Pid ! M1.               % Forward the first received

c(Pid) ->
    receive
        M -> Pid ! M    % Forward the message
    end.
```

Figure 5.1: Nondeterministic ordering of two messages

Furthermore, consider the changes in Figure 5.2 to the previous program (process `C` remains as before). The possible orderings in which the (now three) messages can be delivered at `B` are, in order of likelihood: `t1`, `t2`, `t3`; `t1`, `t3`, `t2`; and `t3`, `t1`, `t2`. Since `t1` and `t2` have the same sender and destination processes, their relative order is fixed. Obviously, it is necessary that the sending of `t2` can be completed even if `t1` has not yet been delivered, if the model is to describe all possibilities allowed by the language semantics.

Since we assumed that the only source of nondeterminism in the language is the choice of sender and receiver in synchronizations, the nondeterministic selection of the next message to be delivered must be modelled by processes performing *send* operations over the same channel. This implies that carrier processes, once handled a message, will be unconditionally suspended until the message has been delivered. Thus, the last example shows that apart from a carrier process, at least one other process is necessary for each source–destination pair.

Letting this other process be an (unbounded) buffer server, acting on requests from the sender and carrier processes (see [17] for an early example of a bounded buffer process using synchronous communication), our model of the asynchronous messaging system is complete, as illustrated in Figure 5.3. The full model of ERLANG inter-process communication is described in more detail below.

5.1.3 Timeout

The timout mechanism can be easily modelled in the following way: a separate (externally defined) timer process is spawned by the `receive`, whose task it is to

```
-module(nondet_2).
-export([a/0]).

a() ->
    A = self(),
    B = spawn(nondet_2, b, [A]),
    C = spawn(nondet, c, [B]),
    B ! t1,
    B ! t2,                 % Send a second message to B
    C ! t3,
    receive M -> M end.

b(Pid) ->
    receive M1 -> M1 end,       % Receive three messages
    receive M2 -> M2 end,
    receive M3 -> M3 end,
    Pid ! M2.                   % Forward the second
```

Figure 5.2: Nondeterminism and relative ordering of messages

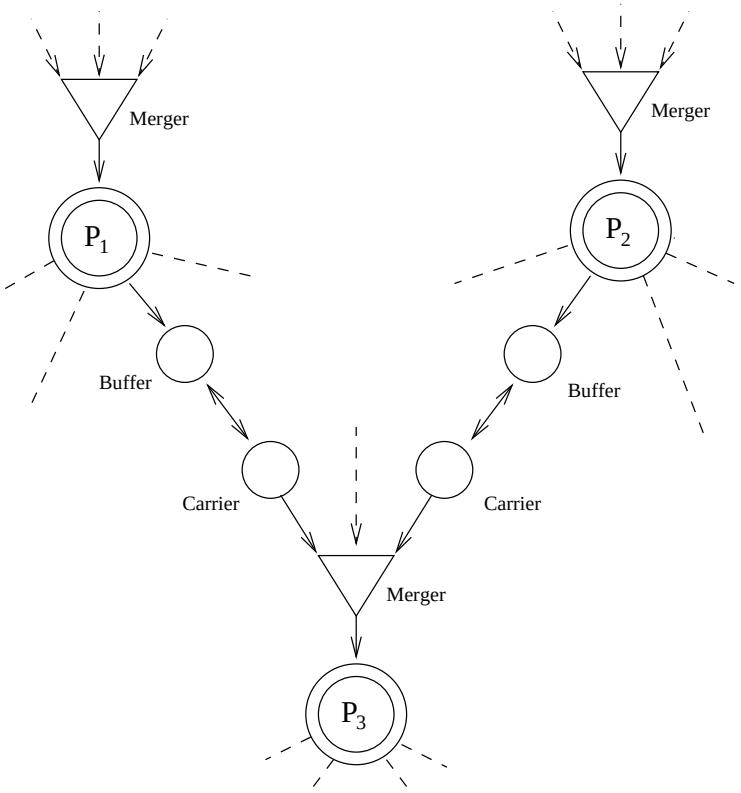

Figure 5.3: A synchronous model of the ERLANG messaging system

$$
\begin{aligned}
&!\,(\pi, M) = \\
&\quad c = \mathit{bufferchan}(\mathit{self}(), \pi), \\
&\quad \mathit{send}(c, \texttt{write}(M)), \\
&\quad M
\end{aligned}
$$

Figure 5.4: Asynchronous send ('!')

send a timeout message when the specified period has passed. This message may not be detected immediately by the receiving process, but will eventually (like all messages) be delivered. Thus it cannot be assumed that no other message will be received after the specified time has elapsed; it only guarantees that the process executing the `receive` will not suspend indefinitely. This follows the Strand implementation [1] of ERLANG, and should be true for the language in general.

We want to keep the protocol between the receiver and the timer processes minimal, so we do not include any acknowledging of messages sent between them. Therefore, when a `receive` which has spawned a timer finds a matching (normal) message, it cannot know whether the timer process is still waiting, or has dispatched a timeout message and terminated. In order to eliminate false timeouts, each new timer and the `receive` call which spawned it must therefore be associated with a unique key, to be used to authenticate the timeout message. Timeout messages whose keys do not match that of the active `receive` call may then simply be discarded.

5.1.4 The complete model

We give here a full description of a synchronous model of the inter-process communication in ERLANG, using a pseudo-functional notation with semantics similar to ERLANG itself; in particular, ',' (comma) denotes the sequencing operator. While the model has not been formally verified, it should be regarded as nothing more than a sketch.

With each ERLANG process, we associate a channel for synchronous communication, which is identified throughout the system via the Pid of the owner process, and let the function $\mathit{input}(\pi)$ yield the channel identified by Pid π.

Asynchronous Send

The definition of the '!' (asynchronous send) primitive is given in Figure 5.4, in accordance with the discussion in section 5.1.2. The call $\mathit{bufferchan}(\pi_1, \pi_2)$ yields the channel for sending to the process which is buffering messages from the ERLANG process with Pid π_1 to that with Pid π_2. We may assume that if such a buffer process did not exist previous to the call then it will be spawned, together with a carrier process, both given in Figure 5.5. These processes must remain active at least until either the sender has terminated and no messages remain to be delivered, or the receiver terminates; for clarity, however, we leave out such details from the model. *self*(), as before, yields the Pid of the executing

$$
\begin{array}{l}
\mathit{carrier}(b, c, d) = \\
\quad \mathit{send}(b, \texttt{read}), \\
\quad \texttt{msg}(M) = \mathit{receive}(c), \\
\quad \mathit{send}(d, \texttt{msg}(M)), \\
\quad \mathit{carrier}(b, c, d)
\end{array}
$$

$$
\begin{array}{l}
\mathit{buffer}(c, d, Q) = \\
\quad x = \mathit{receive}(c), \\
\quad \left\{ \begin{array}{ll}
\mathit{buffer}(c, d, \mathit{append}(M, Q)) & \text{if } x = \texttt{write}(M) \\
\mathit{transfer}(c, d, Q) & \text{if } x = \texttt{read}
\end{array} \right.
\end{array}
$$

where $\mathit{transfer}(c, d, Q) =$

$$
\left\{ \begin{array}{ll}
\begin{array}{l} \texttt{write}(M) = \mathit{receive}(c), \\ \mathit{send}(d, \texttt{msg}(M)), \\ \mathit{buffer}(c, d, Q) \end{array} & \text{if } Q = \texttt{[]} \\
\begin{array}{l} \mathit{send}(d, \texttt{msg}(\mathit{head}(Q))), \\ \mathit{buffer}(c, d, \mathit{tail}(Q)) \end{array} & \text{otherwise}
\end{array} \right.
$$

Figure 5.5: The carrier and buffer server processes

ERLANG process, $\mathit{append}(x, L)$ appends element x to list L, and $\mathit{head}(L)$ and $\mathit{tail}(L)$ select the head and tail parts respectively of list L.

Asynchronous Receive

The model of the asynchronous `receive` primitive is divided into two stages: the initial stage where the buffer holding already delivered messages is searched, and a waiting stage where new messages are accepted one at a time, until a match is found or a timeout occurs.

For the following, we have assumed the existence of a primitive $\mathit{match}(\Phi, x)$, where Φ is a sequence $\phi_1, \ldots \phi_n$ of patterns (with guards), which yields an integer $i > 0$ if ϕ_i is the first pattern in Φ that is valid for x, or zero if no match is found. The representation of patterns and guards is left out of this discussion.

We have also assumed that each ERLANG process has an associated buffer of terms stored in incoming order, with the operators $\mathit{store}(x)$, which adds a term x to the buffer, $\mathit{delete}(x)$, which removes the oldest occurrence of term x from the buffer, and $\mathit{lookup}(\Phi)$, which yields $\texttt{msg}(x)$ if x is the first term in the buffer such that $\mathit{match}(\Phi, x) > 0$, or the unit tuple () if no such x exists,

Let `receive` be defined as shown in Figure 5.6, where the parameter Φ is a sequence $\phi_1, \ldots \phi_n$ of patterns, and $E = e_1, \ldots e_n$ a sequence of corresponding expressions, for any number $n > 0$. The parameter t is a timeout limit (which is either a nonnegative integer or `infinity`) and T is the corresponding expression. `case` is simply the ERLANG case primitive, taking the obvious parameters.

The primitive $\mathit{timer}(t, c)$ is assumed to spawn a timer process, given a positive

$\texttt{receive}(\Phi, E, t, T) =$

$$x = \mathit{search}(\mathit{input}(\mathit{self}()), \Phi, t),$$
$$\begin{cases} T & \text{if } x = () \\ \texttt{case}(M, \Phi, E) & \text{if } x = \texttt{msg}(M) \end{cases}$$

where $\mathit{search}(c, \Phi, t) =$

$$x = \mathit{lookup}(\Phi),$$
$$\begin{cases} \mathit{delete}(M), x & \text{if } x = \texttt{msg}(M) \\ () & \text{if } x = () \text{ and } t = 0 \\ \mathit{waiting}(c, \Phi, ()) & \text{if } x = () \text{ and } t = \texttt{infinity} \\ \pi = \mathit{timer}(t, c), & \\ \mathit{waiting}(c, \Phi, \pi) & \text{otherwise} \end{cases}$$

and $\mathit{waiting}(c, \Phi, \pi) =$

$$x = \mathit{receive}(c),$$
$$\begin{cases} () & \text{if } x = \pi \text{ and } \pi \neq () \\ x & \text{if } x = \texttt{msg}(M) \text{ and } \mathit{match}(\Phi, M) > 0 \\ \mathit{store}(M), & \text{if } x = \texttt{msg}(M) \text{ and} \\ \mathit{waiting}(c, \Phi, \pi) & \mathit{match}(\Phi, M) = 0 \\ \mathit{waiting}(c, \Phi, \pi) & \text{otherwise} \end{cases}$$

Figure 5.6: Asynchronous `receive`

integer t and a channel c for synchronous communication, and yield the (unique) Pid of the new timer as its return value. Following the discussion in section 5.1.3, we use this Pid to authenticate the timeout message. The timer will then simply send its own Pid on channel c after time t (measured in milliseconds), and then terminate.

5.1.5 Summary

Expressing the semantics of ERLANG message passing in terms of synchronous communication primitives is a more complicated matter than it would appear at a first glance. Apart from the processes executing the ERLANG code, there must be an extensive structure of buffer and carrier processes – one of each per pair of sender and receiver processes.

The model as detailed above is interesting in its own right, while it makes apparent several aspects of the ERLANG semantics. It should be obvious, though, that it is inadequate as a direct means to facilitate communication analyses, mainly because of the combinatorial explosion in the state space, caused by the extra processes, the buffering, and the nondeterminism.

Analyses for systems using synchronous communication all exploit the fact that such systems are (generally) constructed so that there is a tight coupling between the producing/sending and the receiving/using of data – processes in such systems often proceed in lock-step. ERLANG programs, however, are not typically written in this way, because the asynchronous messaging system encourages the programmer to write more "relaxed" code.

Apparently, on top of the above model is required an abstraction which treats the message buffering processes separately from the processes executing ERLANG code, and so nothing is gained from basing an analysis on a synchronous model. However, where a communication analysis can decide that an ERLANG program displays synchronous behaviour, methods from that area may be successfully applied.

5.2 Concurrent Constraint Programming

Concurrent Constraint Programming, or as it is often known, Concurrent Logic Programming, is basically logic programming extended with (or rather, generalised to) concurrency. Since there is much natural parallellism in most logic programs, this is quite a straightforward generalization – much more so than the concurrent extensions of imperative or functional languages. In the latter, concurrency is primarily explicit[1], i. e., separate processes are created only through execution of instructions such as the `spawn` in ERLANG. In Concurrent Constraint Programming, on the other hand, *all* computations are implicitly concurrent, and instead, any data dependencies must be given explicitly. The closeness to logic programming and mathematical logic has made it an attractive field of research, and much work has been done in this area in the last years regarding static analysis of data dependencies.

[1] Explicit at least for communicating processes. Isolated subcomputations may be executed concurrently in so-called *threads*, and this is sometimes done implicitly.

5.2.1 Introduction to CCP

From a simplistic view, the Concurrent Constraint Programming, or CCP, paradigm reduces all computational concepts to the execution of a simplest form of processes usually called "atoms". The term "constraint programming" refers to the view of variable bindings as a global, monotonically growing set of constraints on variables. A process (an atom) executes atomically once scheduled, terminating in constant (and very short) time. The actions an atom can perform are limited to the instantiation of variables and the spawning of new atoms. (The parent atom always terminates before any of its children are scheduled.) Atom execution is usually referred to as *reduction*, and is performed much as a function call in ERLANG, with the modification that *any* matching clause may be selected – not necessarily the topmost. We say that a computational state σ_i reduces to state σ_j with selected atom A (i. e., the atom reduced at that step) in σ_j, and selected clause C in the program, and write $\sigma_i \longrightarrow \sigma_j$. Each atom spawned in a reduction is considered distinct from all atoms occurring in some previous state (and all variables introduced in the selected clause are renamed away from any previously occurring variables).

The main difference from ERLANG and other functional languages is that in CCP, variables can be passed around without being bound. Instead of calling a function and wait for it to return a value, an unbound variable can be passed to the subcomputation, as a slot to place the result in. (Any number of unbound variables can be passed, so the code for an atom does often not denote a unique mathematical function). Unbound variables can also be made part of data structures, which are then referred to as *incomplete*.

Data dependencies

Synchronization in CCP is handled solely by use of clause guards, demanding that certain input variables be bound (at least partially) to certain values before the clause may be selected. An atom remains suspended until some clause matches (or until no clause is consistent with the variable bindings, in which case the atom fails). Thus, an atom can be made to wait until another atom instantiates a variable shared between them. Commonly, CCP languages are so-called *committed choice* languages, implying that there is no backtracking, and consequently, atom failure either causes the program execution to fail (being the "proper" action), or suspends the atom forever (a robust solution used e. g. by Strand, but less correct).

In the most general, still monotonic form of CCP, a program clause (or *rule*) can be written on the following form (see e. g. Yardeni *et al.* 1990 [25]):

```
Head <- Ask : Tell | Body
```

where `Head` is the clause head as in ERLANG, `Ask` is a sequence of guards, `Tell` is a sequence of variable bindings and `Body` a sequence of atoms to be spawned. The clause cannot be selected unless the `Tell` bindings are consistent with the current state, and if it is selected, the bindings are performed as part of the (atomic) reduction. In languages like Strand [15], there is no `Tell` part, and bindings are instead performed by externally defined atoms included in the body. This does in fact reduce their expressiveness somewhat; see [25].

Lastly, unlike ERLANG, where the comma-separated expressions in a clause body are evaluated in textual order, those atoms spawned by a reduction in

CCP may be scheduled in any order, unless they explicitly synchronise via shared variables, as described. In other words, the sequence of atoms in the body of a CCP clause should be regarded as unordered. (Generally, this is also true for the `Ask` and `Tell` sequences, but e. g. Strand guarantees left-to-right execution of the guard tests.)

5.2.2 Translating Erlang into CCP

Leaving out the handling of process failure and error recovery, system calls, *etc.*, there are three primary problems to be addressed in translating an ERLANG program into CCP code. (A detailed description of the translation into Strand is given by Armstrong and Virding in [1].) The first problem is to implement the functional semantics of ERLANG; this is simply handled by adding an extra argument to the ERLANG clauses, to hold the result. The second problem is to sequentialise the execution of the calls in the ERLANG clause bodies. This can be done by chaining a variable through all atoms, using two extra arguments – one to be waited on until it becomes bound, and another which is bound to the value of the first by the execution of the atom. (The actual value bound to the variables is not important to the method, but typically the current process state information would be passed along this chain.)

The third primary problem is to make sure that the clauses defining a function are tried in textual order. This however calls for a nonstandard extension to CCP, such as the `otherwise` guard test found in FCP(:,?) [25] and Strand, and can not be implemented otherwise.

Translating the `case` construct (including the simpler `if` form – see section 4.1) is straightforward, since as mentioned earlier it can be seen as shorthand for a call to an anonymous function, and all occurrences may simply be transformed into calls to unique "lifted" functions. E. g.,

```
f(X) ->
    if
        X == 0 -> 1;
        X >  0 -> X * f(X - 1)
    end.
```

can be rewritten as

```
f(X) -> f_case_1(X).

f_case_1(X1) when X1 == 0 -> 1;
f_case_1(X1) when X1 >  0 -> X1 * f(X1 - 1).
```

Message streams

The only real problem left to do with the ERLANG semantics is then that of message passing between processes. First of all, we need to define what an ERLANG process corresponds to in an execution of the translated program.

An atom A occurring in one or more states in a reduction sequence $\sigma_1 \longrightarrow \sigma_2 \longrightarrow \cdots$ is a *descendant* of an atom B in the sequence if and only if A was spawned as the result of reducing B, or reducing a descendant of B.

An ERLANG process, then, can be identified with the set of all atoms in the sequence that are descendants of some atom $A = \mathit{spawn}(\ldots)$, and which are

```
write(Pid, M, X, X1) <- X = [{Q, M1} | Xs] : true |
                        write(Pid, M, Xs, X1).
write(Pid, M, X, X1) <- true : X = [{Pid, M} | X1] |
                        true.
```

Figure 5.7: Writing to a shared stream

not descendants of a spawn that is a descendant of A. (For simplicity, we can assume that the initial state contains exactly one atom, which is a spawn.)

It is not obvious how shared streams with an unbounded and arbitrary number of producers (so-called *multiple writer streams*), like the message streams of the ERLANG semantics, can be implemented in CCP – at least not without resorting to built-in extensions in the target language (thus removing all reasons for attempting to use a CCP model for analysing the communication in ERLANG programs): the Strand implementation used built-in stream merger operators.

It turns out that the problem is equivalent to that of assigning each producer a unique ID[2]. Given a unique identifier, each producer can write to the shared stream using the definition in figure 5.7, where `X1` should be unbound before the call, and afterwards replaces `X` as the stream reference. The unique Pid guarantees that no tuple {`Pid, M`} already exists in `X`. (A last stage can easily be added between writers and consumers which transforms the stream `[{P1, M1}, {P2, M2}, ...]` into `[M1, M2, ...]`.) Note that `Q` and `M1` are merely placeholders in the above, and that `Q` is never equal to `Pid` if the first clause matches.

(Also note that a direct Strand implementation would have to use a "trick" in the second clause, requiring that the head of the stream be unbound, since Strand clauses have no `Tell` part. Such tests make a program lose certain mathematical properties, e. g. stability of non-delay, which are central to much of the reasoning about CCP programs; in fact, such properties are one of the main things that make CCP interesting, it being rather different from concurrent imperative or functional languages. This point is not of importance to us, though, since we are not restricted to the Strand subset of CCP.)

Reversely, given multiple writer streams, a separate process can be created to receive requests for unique identifiers, and hand them out one at a time, e. g. in an unbound variable sent as part of the request. One consequence of all this is that the mechanism for assigning Pids does not need to be explicitly specified in the translation, but can be invisibly incorporated in the execution of spawns, knowing that it does not violate the semantics of monotonic CCP.

Stream mergers

As it turns out, in a generic CCP model of ERLANG, a sending process will not directly write the message into the input stream of the receiver. As the discussion in section 5.1.2 showed, to cover all possible interpretations of the ERLANG semantics, there must be a separate stream between each pair of sender

[2] Unique process identifiers can be implemented directly in a fairly straightforward manner, using strings such that Pid π of process p is a proper prefix of Pid π' of process q if and only if q is a descendant of p, and such that no two processes spawned from the same parent have the same Pid. (Using binary strings, the child's Pid can e. g. be formed from that of the parent by appending one zero for each previous spawn performed by the parent, followed by a one.)

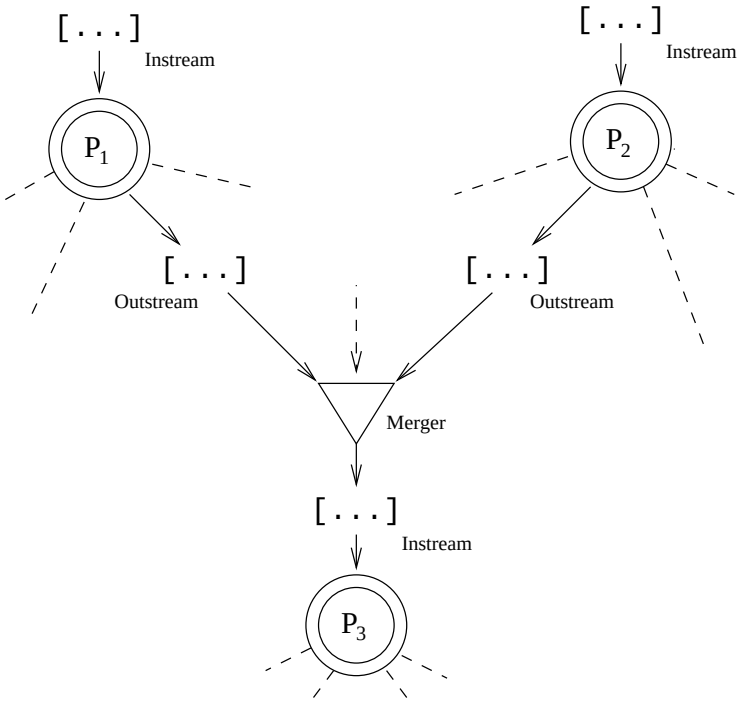

Figure 5.8: A CCP model of the ERLANG messaging system

and receiver processes, and the input stream of a single ERLANG process must be described as the nondeterministic merge of all its incoming message streams. (In the Strand implementation, all outgoing messages – and most system requests – were written to a single out-stream read by the kernel process, which in its turn distributed them to their respective destinations, but such a centralised model is not general enough for our purposes, and would unnecessarily complicate program analysis.) The generalised structure of processes and streams in a CCP model is shown in Figure 5.8. (Note the close similarity to Figure 5.3.)

A generic stream merger can be written in such a way that for each in-stream that is to be included in the merge, a separate sub-process is spawned which takes one message at a time from the in-stream and tries to write it to the shared outgoing stream, similarly to the carrier processes of section 5.1.4. (An order to add or remove a stream can be passed as a message in any in-stream already being read, and acted upon by a filtering stage in the merger.)

However, a merger for ERLANG message streams should be fair, in the sense that in any state, the probability should be equal for all in-streams containing accessible messages, that the next message written to the out-stream is read from any particular one of them. A merger constructed using the write operation defined in figure 5.7 will not satisfy this requirement, even if the scheduling of processes (atoms) in itself is fair. To see this, consider a case where two in-streams, of which at least one is infinite, are being merged via two writer processes by use of this definition. There is then at least one fair, infinite, reduction sequence (namely, that which alternates between the two writer processes)

where one infinite stream is copied to the out-stream, while the other stream is blocked indefinitely because its writer process is "wasting" its reductions trying to find an empty slot for the next message. For instance, in [21] Shapiro and Safra use a built-in operator, which places a message in the first uninstantiated slot of a stream as an atomic operation, to implement a fair multiway merge. It would seem that such a merger cannot be achieved in pure monotonic CCP without such built-in extensions, but since the set of all fair mergings is a subset of all possible mergings, as expressed by the above model, any analysis using it as a model of message stream merging in ERLANG would still be conservative (safe).

Reading from a stream

The procedure for receiving a message from the input stream of an ERLANG process can actually be expressed somewhat simpler in CCP than in imperative or functional concurrent programming languages (such as Concurrent C and Concurrent ML), because it does not have to be divided into the two stages of first searching the buffer, and then attempting to receive further messages, should the first stage fail. (The handling of timeouts remains equivalent to that described in section 5.1.3.)

Since the input stream is an incomplete data structure – a list whose tail remains partially undefined until the stream is closed – the buffering of unconsumed messages can be handled simply by not removing them from the stream. Each attempt to receive a message will search the present stream from the beginning, not making any difference between messages that were accessible during a previous search, and messages that have appeared since then. If no match is found before the unbound part of the stream is reached, the search will simply suspend until more of the stream becomes defined. When a matching message is found (be it a timeout message or otherwise), the new stream used for later searches will be that composed of (a copy of) the old stream up to but not including the matched message, followed by that (actual) part of the old stream which followed the message.

5.2.3 Conclusions

It is apparent that a CCP model of the sequential execution of ERLANG processes is not helpful for analysis purposes; if anything, the introduced execution order dependencies between atoms will blur distinctions between any abstract state descriptions. Nor are the actual data dependencies via the message streams made more obvious in a CCP model than in e. g. the synchronous model described in section 5.1.

What has been shown here of main interest are the difficulties in modeling the merging of ERLANG message streams in monotonic CCP, and how the structure of streams and merger processes is laid out. We note that the latter is quite similar to how queues of message carrying processes are formed in the synchronous model, and conclude that an abstraction of the message passing in ERLANG should be based on this structure.

5.3 Summary

We have expressed the semantics of the ERLANG messaging system in the concurrency models of synchronously communicating sequential processes (with dynamic creation of processes and channels) and Concurrent Constraint Programming, demonstrating that neither is a useful basis for the analysis of ERLANG programs. However, the construction of both the respective models has showed upon several interesting details regarding the semantics of ERLANG message passing, and has pointed out the common basic structure of the messaging system in its most general form, as shown by figures 5.8 and 5.3.

Chapter 6

Deadlock analysis of Erlang programs

We have identified only two existing analyses of synchronization and communication as being powerful enough for the purposes we specified in section 1.2, and specifically, for the verification of absence of deadlocks in concurrent ERLANG programs. These are, respectively, the synchronization and aliasing analysis of Colby [8], and Masticola's method of locating and eliminating possible dependency cycles [19]. We described the former in section 1.6, and, although we deem that it would be quite interesting to see it applied to ERLANG programs, its apparent computational expensiveness and the lack of practical test results so far has prompted us here to focus instead on the latter method, which we believe could give results that satisfy our requirements – in particular if combined with the modular verification strategy suggested in section 1.5.

6.1 Masticola's method

The intuitive idea behind this method of verifying absence of deadlocks in a program, is that corresponding to any deadlock (but not to pure cases of starvation) there is a circular *waits-on* dependency over the involved processes. If we can build a dependency graph over the program, which contains at least all such dependencies between program points that are actually possible, and then show that no dependency cycle in the graph (there will usually be many) can exist in an actual execution of the program, we will have shown that the program cannot deadlock. (Although the possibility remains that its processes can become starved.)

The synch graph

To do this, Masticola lets the flow graph over the program text represent all possible machine states (sets of processes with current program points in the graph). This graph is abstracted away from instructions other than those of synchronization, so that a node in the graph actually represents all execution path segments from previous synchronization instructions to that particular instruction, which do not pass through another synchronization instruction.

Directed *synch edges* are then added to the graph, from sender S to receiver R, for all program points S, R such that it may be possible for a process at S to enable a process concurrently at R to proceed. Since the approximation must be safe, a synch edge can only be left out where it can be shown that such a synchronization *cannot* occur. For synchronization schemes such as rendezvous, synch edges may be bidirected or undirected. We further discuss how the set of synch edges is determined in section 6.2.

In its basic form, then, no distinction is made between process instances when this extended *synch graph* is built. However, to handle programs with dynamic process creation, Masticola suggests that the relevant parts of the program text be duplicated, possibly several times, so that different classes of process instances may be described by separate parts of the synch graph. This method, applied by Masticola to Concurrent C programs, appears to be quite limited, though, and tends to need the programmer's help in deciding how to suitably partition the processes.

Deadlock cycles

Any possible waits-on dependency between two processes of the program is now represented by a path from the waited-on node to the waiting node, consisting of zero or more control flow edges followed by exactly one synch edge. The path represents the possibility that if the waited-on process is able to continue execution, it may reach a program point where it can enable the waiting process to proceed. Obviously, both nodes must represent synchronization instructions that may suspend process execution, and it must be possible for these program points to be executed by distinct processes, or no actual dependency can exist (if we assume that no process can be dependent on itself).

Intuitively, a deadlock in an execution of the program must be represented by two or more such waits-on dependency path segments, connected so as to form at least one cycle. (There are problems with formalising this intuition, which we discuss in section 6.2.) The algorithm for verifying absence of deadlock, then, is to locate every such cycle in the synch graph, and show for each found cycle – using any available additional information about the program – that it cannot represent a deadlock in an actual execution of the program.

Apart from the previously mentioned necessary characteristics of these so-called *deadlock cycles*, Masticola identifies the following:

- There must be a reachable program state in which the waited-on program points are executed concurrently by distinct processes.
- There must be a reachable program state such that the waited-on nodes in the cycle are all simultaneously unable to proceed, given the current state of the program store, signals, messages, *etc.*
- In any program state for which the above criteria hold, no process in the system must necessarily be able to enable a process within the cycle to proceed (breaking the cyclic wait).

If any of these constraints can be shown not to hold for a cycle in the graph, that cycle cannot represent a possible deadlock of the program. (The last constraint is generally very difficult to prove false, but it can be done in special cases.)

Cycle pruning information

The kind of information necessary to efficiently prune possible deadlock cycles generally involves the particularities of the semantics of synchronization in the source language, but most important is the estimation of a *Can't Happen Together* relation (Masticola's term) which describes the non-concurrency of pairs (or sets) of process states. It may for instance be possible to decide, for processes p and q, that p is never executing a suspending synchronization instruction when q is, and thus no actual deadlock cycle can contain both p and q. The details of *CHT* analysis are complicated – an important ingredient is a previous stage of analysis determining a "must have completed before"-relation on program points – see [19] for mathematical models and examples of analyses of Ada, binary semaphores, and Concurrent C.

6.2 Our extension

Masticola however fails to recognise that the concept of a "waits-on" dependency between processes is more complicated than our first intuition gives at hand. In the case of Ada programs, senders and receivers are statically and explicitly matched by the program text, and the synch graph can be built during parsing. The synch edge representation of possible synchronizations is obviously safe in this case – we can be sure that all possible dependencies have been included by the construction algorithm – but it is not obvious how this process can be generalised to other languages and concurrency models.

A program verified free from deadlock cycles with Masticola's method is not necessarily free from cases of starvation, and whether a process is regarded as starved or not depends on the rule used for determining possible process dependencies. If this rule is too conservative, there may be cases of indefinitely suspended processes which we would want to describe as deadlocked, but which will not be regarded as cyclically dependent under the rule in question. On the other hand, if the rule includes unnecessarily many suspected dependencies, it will be very difficult to show for all ensuing cycles that they cannot correspond to actual deadlocks.

It is also far from obvious what the absence of deadlock cycles implies regarding the existence of local deadlocks (livelocks) – particularly in the case when processes can be dynamically created by the program. In [4], we give a thorough treatment of all the above questions, defining the class of *weak process dependency relations*, together with a basic set of rules for determining when such dependencies can safely be excluded between pairs of processes, and showing how Masticola's method can be extended using these dependency relations to verify absence of local deadlock. In particular, we show under which conditions absence of deadlock cycles implies absence of local deadlocks, even for programs with dynamic process creation.

6.3 Precision problems

In applying this extended method to ERLANG programs, there are some problems with the precision of the analysis that do not occur for programs in Ada (although some do for Concurrent C). The probably most important of these is

the difficulty in matching sender and receiver processes. Not only do ERLANG programs use dynamically created channels (one for each new process), and use local variables to hold the process identifiers (Pids) that are used as channel designators in send ('!') instructions, but they also regularly pass Pids between processes (often referred to as *channel migration*), and in several applications store Pids deeply embedded in large data structures. This is not an insurmountable problem, though – recently, Sven-Olof Nyström has achieved promising results with such a *communication topology* analysis even for large ERLANG programs (personal communication, March 1997).

The problem of finding a generic approach to handling dynamic creation of processes is less straightforward. The subgraph duplication method applied by Masticola for analysing Concurrent C programs requires a good characteristic for separating process instances. (A simple example would be by the values of the initial arguments to the process.) Masticola suggests using the chain of process creation instructions that precede a particular process creation to separate instances, but this has the drawback that the synch graph is then no longer polynomially bounded in size. Colby's trace-based analysis of the communication topology [9] of concurrent programs seems to be a good candidate for improving this situation.

Lastly, the need for a representation in any analysis of the contents of message buffers in the asynchronous inter-process communication system of ERLANG (see chapter 5) is likely to blur many data dependencies. There is however some statistical indication that in a majority of the synchronizations in actual ERLANG programs, the message buffer of the receiving process is empty when the `receive` statement is executed, and also that many synchronizations in practice display synchronous behaviour. An analysis which can identify such cases could probably be used to both simplify the analysis and increase its precision.

6.4 Conclusions

Cycle detection and elimination, as opposed to most other suggested methods for verifying absence of deadlocks, has been shown (by Masticola) to be applicable in practice to large real-world programs, and with quite good results (see [19]). It also has the advantage over most other methods that it can locate possible local deadlocks as well as global, and give meaningful information about their causes. The method is shown fast enough to be practical, and generally has better time behaviour than other analyses. (See section 1.6 for related work.)

We believe that our extension of Masticola's method, described in detail in [4], could give useful results if applied to ERLANG programs, given that the precision problems described above can be solved satisfactorily, and we hope that in the future, an attempt will be made to do this.

Bibliography

[1] Joe Armstrong and Robert Virding. Programming telephony. In *Strand: New Concepts in Parallel Programming, by Ian Foster and Stephen Taylor*, pages 289–304. Prentice Hall, 1990.

[2] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.

[3] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[4] Richard Carlsson and Håkan Millroth. On cyclic process dependencies and the verification of absence of deadlocks in reactive systems. Computing Science Department, Uppsala University, February 1997.

[5] Shing Chi Cheung and Jeff Kramer. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593, August 1994.

[6] Michael Codish, Moreno Falaschi, and Kim Marriott. Suspension analyses for Concurrent Logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.

[7] Michael Codish, Moreno Falaschi, Kim Marriott, and William Winsborough. Efficient analysis of Concurrent Constraint Logic programs. In *Proceedings of the 20th International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science, Vol. 700*, pages 633–644. Springer-Verlag, Berlin, 1993.

[8] Christopher Colby. Analysis of synchronization and aliasing with abstract interpretation. Unpublished.

[9] Christopher Colby. Analyzing the communication topology of concurrent programs. In *The ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 202–214, 1995.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[11] Saumya Debray, David Gudeman, and Peter Bigot. Detection and optimization of suspension-free Logic Programs. *Journal of Logic Programming*, 29(1–3):171–194, 1996.

[12] Saumya K. Debray. Efficient dataflow analysis of logic programs. *Journal of the ACM*, 39(4):949–984, 1994.

[13] Alain Deutsch. *Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data*. PhD thesis, LIX, Ecole Polytechnique, Palaiseau, France, 1992.

[14] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE 1992 International Conference on Computer Languages*, pages 2–13, San Francisco, California, April 1992.

[15] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming*. Prentice Hall, 1990.

[16] S. Gregory. *Parallel Logic Programming in PARLOG*. Addison-Wesley, 1987.

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

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[19] Stephen P. Masticola. *Static Detection of Deadlocks in Polynomial Time*. PhD thesis, New Brunswick Rutgers, State University of New Jersey, May 1993.

[20] S. G. Matthews. An extensional treatment of lazy data flow deadlocks. *Theoretical Computer Science*, 151(1):195–205, 1995.

[21] E. Shapiro and S. Safra. Multiway merge with constant delay in Concurrent Prolog. *New Generation Computing*, 4:211–216, 1986.

[22] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1986.

[23] Scott D. Stoller and Fred B. Schneider. Verifying programs that use causally-ordered message-passing. Department of Computer Science, Cornell University, Ithaca, New York, April 1995.

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

[25] E. Yardeni, S. Kliger, and E. Shapiro. The languages FCP(:) and FCP(:,?). *New Generation Computing*, 7(2):89–107, 1990.