Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

A Debugger for Communicating Scala Objects

Andrew BATE and Gavin LOWE

Department of Computer Science, University of Oxford, Oxford, OX1 3QD, England.

andrew.bate@univ.ox.ac.uk gavin.lowe@cs.ox.ac.uk

Abstract. This paper presents a software tool for visualising and reasoning about the behaviour of message-passing concurrent programs built with the CSO library for the Scala programming language. It describes the models needed to represent the construction of process networks and the runtime behaviour of the resulting program. We detail the manner in which information is extracted from the use of concurrency primitives in order to maintain these models and how these models are diagrammed. Our implementation of dynamic deadlock detection is explained. The tool can produce a sequence diagram of process communications, the communication network depicting the pairs of processes which share a communication channel, and the trees resulting from the composition of processes. Furthermore, it allows for behavioural specifications to be defined and then checked at runtime, and guarantees to detect the illegal usage of concurrency primitives that could otherwise lead to deadlock or data loss. Our implementation imposes only a small overhead on the program under inspection.

Keywords. concurrency, Scala, CSO, deadlock detection, debugging.

Introduction

Concurrent programming is an important paradigm for structuring programs. The concurrent programming model can simplify the design of multi-task systems. Such systems consist of a collection of semi-independent tasks, each with independent data. Identifying and programming each task as a single process is easier than struggling to design and maintain one monolithic process that is responsible for all the tasks [1]. This separation of concerns leads to a natural exploit of the inherent concurrency of the problem.

Message-passing concurrency is arguably the easiest paradigm to program in and understand: the interaction and cooperation between processes is modelled in a very natural way; the paradigm scales well and there exist well-developed semantic models and robust tools for formal verification of design [2]. This latter point is particularly important: to be confident a solution to a problem is correct, formal models are more often required of parallel programs than of their sequential counterparts [1].

However, reasoning about message-passing concurrency is still difficult and requires significant effort on the part of the programmer. Many of the techniques employed in debugging sequential programs do not readily apply to concurrent programs, since the interleavings of actions from different processes are not known externally. For example, allowing each process to print to the console when it enters a particular state is of limited use in debugging because it is not known whether those processes entered those states in the same order as the messages were printed. At present, considerable time is spent attempting to reason about some execution of a concurrent program, with limited guarantees that the information ob-

tained is accurate. Consequentially, large multi-threaded applications have often been discouraged [3].

Communicating Scala Objects (CSO) is an efficient implementation of message-passing concurrency for the Scala programming language [4]. Scala programs run on the Java VM and are bytecode compatible with Java. Our tool seeks to build upon the advantages of message-passing concurrency using CSO and tries to address the concerns outlined above.

In particular, the tool developed extracts information about the trace of communications and the network of processes, and constructs models representing the evolution of the concurrent program over time. It also allows *behavioural specifications* to be defined in terms of trace patterns of special events. These specifications can then be checked at runtime.

The tool provides comprehensive diagrams depicting these models. The *sequence diagram* shows the time-ordered sequence of communications between processes and also depicts the events logged by processes for behavioural specifications. Figure 1 shows an example of this diagram produced by the tool. Modelling the interaction between the running processes can help provide an explanation for the observed behaviour of a particular run of the program. In addition to the diagram, the event trace of all communications and events for behavioural specifications is shown in the lower left-hand column.

The *communication network* is the graph that results from instantiating a node for every process and that connects each pair of processes that share a channel. Figure 2 provides an example of such a network. The process *composition trees* illustrate the syntactic way in which the processes were composed together to form the network; examples of such trees output by the debugger are shown in Figure 3. Together, these diagrams provide both an aide-mémoire for the programmer and assist with debugging. For example, it would be immediately obvious if the network was not as the programmer intended, for example, if the network unexpectedly contained a cycle.

All of these diagrams can form part of the documentation of the program.

The tool also provides a mechanism for the dynamic detection of deadlock in the program under inspection. This mechanism has been optimised so as to only run when the communication graph induced from the processes contains a cycle—a necessary condition of deadlock [2]. Furthermore, this procedure is capable of detecting deadlock in any subnetwork of the communication graph.

The contract of the CSO library restricts the usage of certain components, such as limitations on the sharing of channels between processes. If this contract is broken, then correct functionality is not guaranteed. Our tool performs additional checks to ensure that these conditions are satisfied and outputs detailed advice to the programmer when an illegal usage is found, locating as closely as possible the point in the program where the contract of the CSO library was breached. The tool also provides real-time data about the runtime environment, including the resource usage of the program, helping the user to further distinguish between deadlock and livelock.

The overhead of the debugger to the running program is small. This low overhead is crucial to the usefulness of such a tool: the timing details and interleavings of the actions of the program under inspection should be altered as little as possible, since altering timing details can have a significant impact on whether deadlock will manifest [1].

The tool and its source code are available from the authors.

Outline of the Paper

In Section 1 we introduce the principle constructs of CSO that we use throughout this paper.

Section 2 provides a high-level overview of the design and challenges of the implementation. Section 2.1 describes the process model, which records the composition of processes and communication channels. Section 2.2 details the model of running threads, and in par-

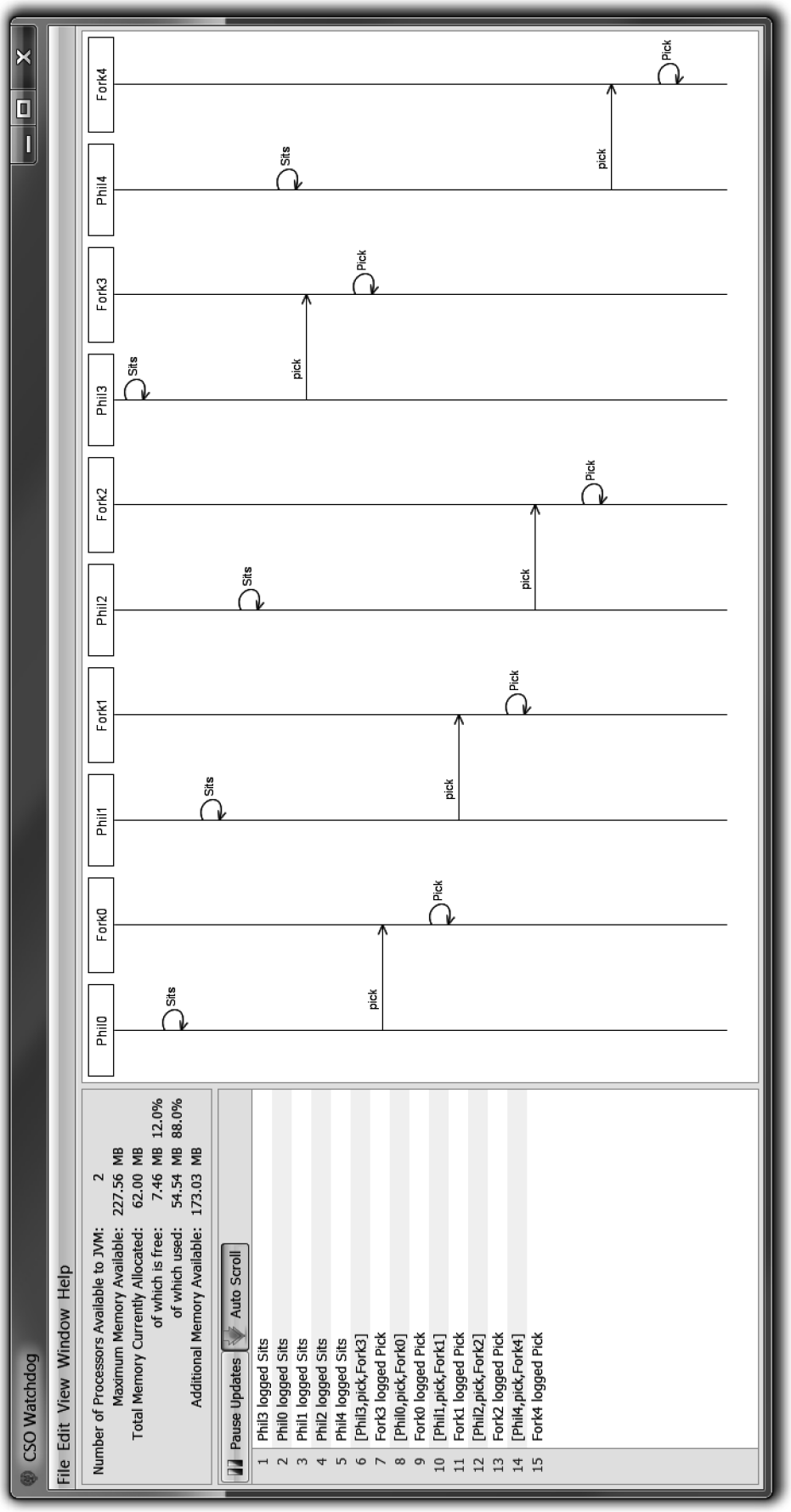


Figure 1. Screenshot of the main screen of the debugger for the Dining Philosophers problem just before deadlock occurs.

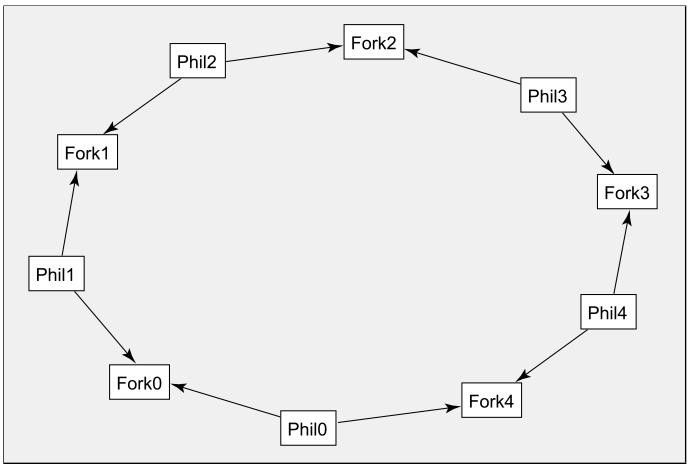
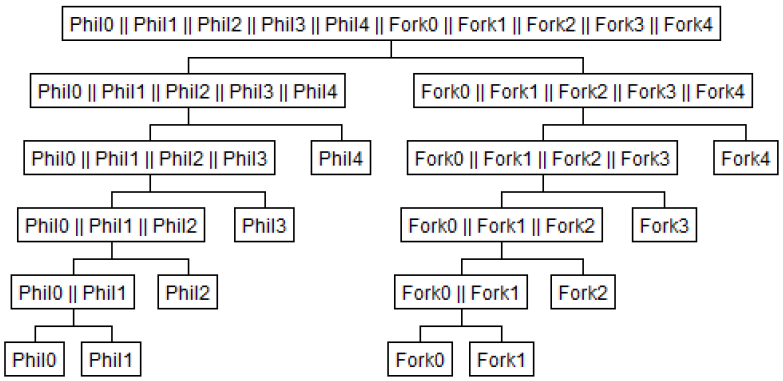
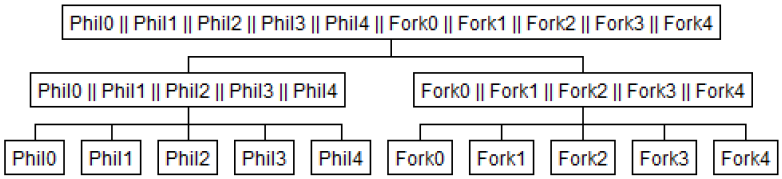


Figure 2. Communication network of the Dining Philosophers problem output by the debugger.



(a) Concrete composition tree



(b) Abstract composition tree

Figure 3. Concrete vs. abstract composition trees for the Dining Philosophers problem produced by our tool.

ticular, Section 2.2.1 describes the mechanism for logging communications. Sections 2.3 explains how behavioural specifications can be defined in the user's code. Section 2.4 details how our deadlock detection algorithm utilises the logged data.

Section 3 offers substantial empirical evidence that the performance overhead of the tool is low. Whilst this is the first presentation of such a comprehensive tool, Section 4 summarises closely related work and compares them to our implementation. We conclude with comments on possible future work.

1. Just Enough CSO

This section introduces the principle constructs of CSO and demonstrates their use with the Dining Philosophers problem.

1.1. Simple Processes

A process is constructed using the notation

```
def ExampleProcess(x1: T1, ..., xn: Tn) = proc (name) { body }
```

where name is a String used to identify the process (when debugging or using our tool), body is a sequence of statements (called the *process body*) to be executed when the process is run, and x1, ..., xn are the arguments to body with the respective types T1, ..., Tn. Any of the Ti may be ports (discussed below in Section 1.3).

If P is a process, then P() runs it in its own thread using a java.lang.Thread.

1.2. Parallel Composition

The parallel composition of processes is achieved using the || operator. For example, the composition of the four processes P1, P2, P3 and P4 is given by

```
val comp = P1 || P2 || P3 || P4
```

and is equivalent to ((P1 || P2) || P3) || P4. A composition of parameterised processes P(0), P(1), ..., P(n−1) can be achieved by[1]

```
|| (for (i <− 0 until n) yield P(i))
```

and results in a process equivalent to ((... (P(0) || P(1)) || ...) || P(n−1)) (i.e. it is implemented using repeated binary composition, associating to the left), where n may be determined at run-time.

If Q is a process representing a parallel composition P1 || P2 || ... || Pn, then Q() causes each of the processes P1, P2, ..., Pn to be run in its own thread. The process Q terminates when all of the components Pi of its composition have terminated.

It is important to reaffirm the distinction between a thread and a process: a *process* is an object of type ox.cso.Process that specifies the runtime behaviour of a thread, whereas a *thread* is a JVM thread running the body of some process.[2] A process may be run by more than one thread. For example, executing the statement (P || P)() results in two threads running separate instances of the body of process P.

1.3. Ports and Channels

Processes communicate via channels. A channel on which values of type T may be read or written is an instance of a subclass of Chan[T] and consists of one input port of type ?[T] and one output port of type ![T]. Processes are parametrised with the ports on which they may read or write data, as described in the previous section.

Input ports and output ports are named from the perspective of the process. Therefore, an output port provides a method !(T): Unit, called a *shriek*, to write values to the channel. Similarly, an input port provides a method ?(): T, called a *query*, to read values from the channel.

Channels are either *synchronous* or *buffered*. For a synchronous channel, termination of the execution of a ! at the output port is synchronised with the termination of the execution of

[1] In Scala, 0 **until** n denotes the range of integers from 0 to n − 1 inclusive.

[2] This is not the usual distinction between a thread and a process, such as that found in operating systems. The definitions here, however, arise from the fact that a CSO process is an abstraction while it is a Java thread that is responsible for the execution on the JVM.

a corresponding ? at the input port. A buffered channel, however, does not synchronise any execution of ! with ? [4].

A OneOne[T] channel is a synchronous channel where no more than one process at a time may access its output port or its input port. Similarly defined are the channels ManyOne[T], OneMany[T], and ManyMany[T], where the naming convention is the restriction on sharing of the output port followed by that of the input port. Each of these channels have the following buffered counterparts: OneOneBuf[T](n), ManyOneBuf[T](n), OneManyBuf[T](n) and Buf[T](n), respectively, where n is the buffer capacity.

A channel is closed by invoking its close method, and an unshared input (output) port may be closed by invoking its closein (closeout) method. Closing a channel (port) enforces the contract that the channel (port) will never be read from or written to again.

1.4. Alternation

An alt consists of a collection of guarded events:

```
alt ( (guard1 &&& port1) −−> { cmd1 }
    | ...
    | (guardn &&& portn) −−> { cmdn }
    )
```

A *guarded event* is a construct for the form (guard &&& port) −−> { cmd }, where guard is a Boolean expression, port is the input port of a channel, and cmd is some statement. A guarded event is *enabled* if port is open and guard evaluates to true, and it is *ready* if the port can be read from at that instant. Guarded events may also use output ports, in which case the syntax is (guard &&& outport) −!−> { cmd }; it is ready if outport is ready to be written to at that instant. An event is *fired* by executing its cmd [4]. A thread executing an alt waits until it finds an event that is enabled and ready, and then fires that event, after which the alt terminates.

1.5. Example: Dining Philosophers

We illustrate the use of the CSO library with the example of the Dining Philosophers problem [5]. The source file begins by importing the package ox.CSO. We can model the basic actions of the philosophers and forks as:

```
def Pause = Thread.sleep(200)
def Sits  = { println("sits"); Pause; }
def Eat   = { println("eats"); Thread.sleep(250) }
def Think = Thread.sleep(Random.nextInt(400))
def Pick  = println("pick")
def Drop  = println("drop")
```

and for simplicity we assume that printing to the console is an atomic action. Now we can define the philosopher process:

```
def Phil(me: Int, left: ![String], right: ![String]) = proc("Phil"+me) {
  repeat {
    Think; Sits;
    right!"pick"; Pause; left!"pick"; Pause; Eat;
    left!"drop"; Pause; right!"drop"; Pause
  }
}
```

where the statement repeat { body } will repeatedly execute the statements of body. A fork is defined as:

```
def Fork(me: Int, left : ?[String], right : ?[String]) = proc("Fork"+me) {
  repeat {
    alt( left -->  { val x = left?; assert(x=="pick"); Pick
                     val y = left?; assert(y=="drop"); Drop }
       | right --> { val x = right?; assert(x=="pick"); Pick
                     val y = right?; assert(y=="drop"); Drop }
       )
  }
}
```

The channels to pick up and drop the forks are just

```
val N = 5
val philToLeftFork, philToRightFork = OneOne[String](N)
```

where N is the number of forks and philosophers. Here, philToLeftFork(i) is the channel from Phil(i) to Fork(i), and philToRightFork(i) is the channel from Phil(i) to Fork((i−1)%N).

The final system is defined by the parallel compositions

```
def AllPhils = || (for (i <- 0 until N) yield Phil(i, philToLeftFork(i), philToRightFork(i)))
def AllForks = || (for (i <- 0 until N) yield Fork(i, philToRightFork((i+1)%N), philToLeftFork(i)))
def System = AllPhils || AllForks
```

and is run by calling System().

2. Design and Implementation

In this section we provide a high-level description of the architecture of the debugger.

The debugger maintains two models: the *process model* and the *thread event model*. The process model records the composition of processes and the channels and alts that processes may use to communicate. The thread event model records the runtime behaviour of each execution of a process body by a thread, logging both attempted and successful communications and the special events used to define behavioural specifications.

Intuitively, program analysis relating to the construction and composition of CSO processes and the sharing of channel references is stored in the process model and may be shared by many running threads. Analysis relating to the execution of the body of a process by a thread and message passing is stored in the thread event model. Sections 2.1 and 2.2, respectively, explain how the data for these models is acquired.

Specification and checking of process behaviour is described in Section 2.3 and the deadlock detection algorithm is described in Section 2.4. The visualisations of these models are described in Section 2.5.

The structure of the debugger is shown in Figure 4. The arrows represent the flow of information from the running processes to the two models of the debugger and then from these models to the debugger instance for visualisation and analysis.

2.1. Process Model

To use the CSO library with debugging enabled, instead of regular CSO, all the user need do is import ox.LoggedCSO instead of ox.CSO. Our implementation subclasses ox.cso.Process in order to make the modifications described below and has been designed so that existing CSO processes (possibly defined in compiled code) can be conveniently transformed (using a wrapper class) into processes that have their state logged to the same level as processes that

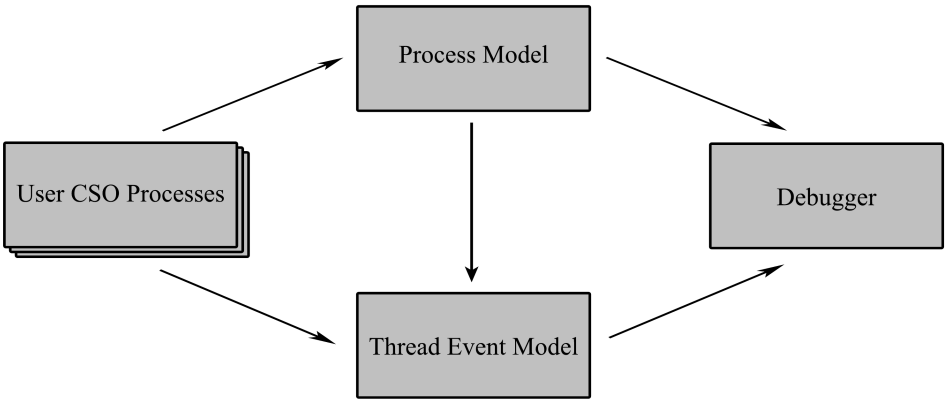


Figure 4. Structure of the Model. Arrows represent the flow of information.

were defined directly using our library. For example, several existing useful components are defined in the ox.cso.Components package.

When a CSO process P is constructed, the process model must be notified of the channel ports connected to P. The ports passed as arguments to a process declaration are obtained using the Java Reflection API. Port references that are not passed as an argument to a process (i.e. those declared in shared memory) cannot be obtained using this API. Therefore, we impose the following syntactic restriction on the way processes are defined in the user's source code:

```
def ProcessName(x1: T1, ..., xn: Tn) = proc (name) { body }
```

and any port used in body must appear as one of the xi whose corresponding Ti is of the form ?[Ti'] or ![Ti']. This ensures that all processes will be logged as having references to exactly the ports on which they may ever communicate.

This stipulation has a secondary benefit: it encourages good design in CSO programs. Therefore we argue that this restriction is entirely a benefit and not an impedance.[3] When it is detected that this syntactic restriction has not been observed (i.e. some process attempts to communicate on a channel that it has not previously registered), or if it is found that a channel has an unused input or output port, the debugger displays a warning to the user which names the offending process definition or channel.

Since port usage is determined at process construction time, and because each process definition may be reused or run concurrently by different threads, using reflection to obtain this data poses little overhead.

The process model also maintains a set of trees of processes. Whenever a process P is constructed, a root node without children is added to the model. If this process P was the result of a binary parallel composition P = L || R, then P becomes a *concrete parent* and L and R are its left and right *concrete children*, respectively. The resulting tree is called a *concrete composition tree*. The concrete composition tree for the Dining Philosophers problem is shown in Figure 3a.

Recall that k-ary parallel composition is implemented via repeated application of the binary parallel composition operator, associating to the left. Therefore, any k-ary composition has a left-skewed binary tree as its model representation. However, in addition to these binary process trees, we should model the construction of the process network as it appears in the source code. Hence, given a k-ary parallel composition of the form

```
val Q = || (for (i <- 0 until n) yield P(i))
```

[3] By this restriction, our tool does not support changing process topology by the communication of ports along channels, known as *mobile channels*. This feature, however, is supported by CSO.

process Q becomes an *abstract parent* and P(0), P(1), ..., P(n−1) are its immediate *abstract children*. This definition of an abstract child augments the existing concrete tree structure. In particular, Q will still have P(0) and P(1) as concrete children (since k-ary parallel composition is implemented using repeated application of the left-associative binary composition operator).

Given a concrete composition tree, its corresponding *abstract composition tree* is defined by the breadth-first traversal starting at the root, and at each node, following only the pointers to abstract children if such children are defined, and otherwise following the pointers to concrete children. Such a tree accurately reflects the programmer's intent and discards the implementation level information. The abstract tree for the Dining Philosophers problem is shown in Figure 3b.

To construct these parallel composition trees, the process model is notified of each parallel composition. Given P = L || R, calling P() causes the body of L to be run in the current thread t and R to be forked off. Hence, although t runs the CSO process P, the behaviour of t is defined by L. Hence binary composition is overridden as follows:

```
override def ||(other: ox.cso.Process): LoggedProcess = {
  // Wrap the result of the superclass in our LoggedProcess subclass
  val result = new LoggedProcess(super.||(other))
  // In the concrete tree: left child is 'this', right child is 'other', parent is 'result'
  ProcessModel.defineConcreteParent(result, this, other)
  // Declare that, when 'result' is run, the code executed in the current thread is defined
  // in the body of 'this' and 'other' is forked
  ProcessModel.redefineProcessImage(result, this)
  result
}
```

Other operators are similarly overridden to log information in the process model. For the case of k-ary parallel composition, we notify the process model of the abstract children that have been defined.

If an alt does not contain an after or an orelse event and no other event ever becomes enabled, then the thread running the alt will wait forever.[4] Hence a process with an alt not containing either an after or an orelse event may participate in a deadlock. Therefore, it is also necessary to record which alt objects a process may execute and the ports used in the events of each alt.

2.1.1. Constructing the Communication Network

The communication network is the graph with one node per process, one node per channel (if any of its ports are shared), and an edge between a process node and a channel node whenever the corresponding process may read from or write to the corresponding channel. This network is constructed using the mapping from processes to ports maintained by the process model.

For example, the program

```
val c = ManyMany[Int]
def Producer(id: Int) = proc("Producer "+id){ repeat { c!id } }
def Consumer(id: Int) = proc("Consumer "+id){ repeat { println(?c) } }
```

[4] If an alt contains an event of the form after(ms) −−> { cmd }, then cmd is executed should no other event fire within ms milliseconds. Similarly, an event of the form orelse −−> { cmd } will cause cmd to execute if no branch is enabled. At most one of after or orelse may be an event of an alt; the debugger guarantees to check this condition, while the native CSO code only provides a partial guarantee. These two events are significant, since both prevent the owner thread of an alt object from deadlocking if no ports ever become ready in the alt.

```
val System = Producer(1) || Producer(2) || Consumer(1) || Consumer(2) || Consumer(3)
```

will result in the communication network shown in Figure 5.

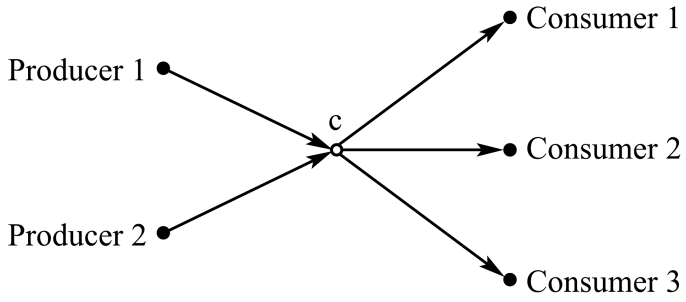


Figure 5. Example of a communication network for a ManyMany channel c.

When drawing communication networks, we construct an edge directly between processes instead of instantiating a node representing the channel for the case of OneOne channels, but nodes for shared channels are included.

2.2. Thread Event Model

This section describes the thread event model, which records the runtime behaviour of each execution of a process body. Section 2.2.1 describes the data that must be logged in order to reason about the inter-process communications.

Whenever a CSO process P is run by the current thread t, the thread event model is notified that t is running the body of P. The model is also notified once the body of P terminates. These two calls to the thread event model allow for the maintenance of a map between CSO processes and the threads currently running them.

2.2.1. Communication Logging

The thread event model is notified before and after every shriek and query. These notifications take the form $\textsc{Before}\,(\Phi, t, c.v)$ and $\textsc{After}\,(\Phi, t, c.v)$ to respectively denote that the operation $\Phi \in \{?, !\}$ is about to be attempted and has just succeeded, for some thread t, some channel c and, if known, a value v.[5]

Attempted operations which have not yet succeeded correspond to ungranted requests, and are used to construct the contention network for deadlock detection (described in Section 2.4). Therefore each $\textsc{Before}\,(\Phi, t, c)$ must be removed from the model on receipt of the corresponding $\textsc{After}\,(\Phi, t, c)$. Successful communications are also drawn on the sequence diagram and reported in the event trace.

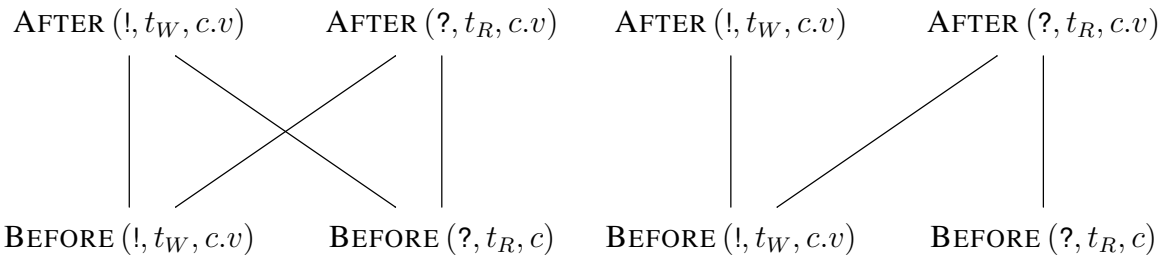
We use Lamport's *happened-before relation* $\preceq$ in order to reason about the interleaving of actions. It is the least strict partial ordering of events such that, for actions a and b, we have $a \preceq b$ only if a is always guaranteed to occur before b [6].

Obviously, for any channel c, for any thread t and any operation $\Phi \in \{?, !\}$ we have that $\textsc{Before}\,(\Phi, t, c) \preceq \textsc{After}\,(\Phi, t, c)$, and since alt objects do not perform channel operations, at most one $\textsc{Before}\,(\Phi, t, c)$ can exist in the model for a given thread t.[6]

[5]For reader threads, the value v communicated along c is not known before the query but is known after. Obviously, for writer threads, the value v communicated along c is known both before and after the shriek. Thus all notifications are of the form diagramed in Figure 6. For conciseness, we elide the value in this notation where necessary.

[6]Recall that the implementation of alt only tests whether a channel is ready for communication; it is not responsible for the communication itself and thus the testing of readiness does not result in a BEFORE notification.

Suppose t_W is a thread performing a write operation on channel c and t_R is the thread that succeeds in performing the corresponding read. Then $\text{BEFORE}\ (!, t_W, c) \preceq \text{AFTER}\ (?, t_R, c)$. If c is a synchronous channel then we are additionally guaranteed that $\text{BEFORE}\ (?, t_R, c) \preceq \text{AFTER}\ (!, t_W, c)$. (For buffered channels, a query attempt may never happen: a thread t' could shriek a value v along buffered channel c with no process t'' ever receiving this value, provided the length of the buffer has not been exceeded.) These orderings are shown in Figure 6.



(a) Synchronous channels. (b) Buffered channels.

Figure 6. Hasse diagrams of strict partial ordering, $\preceq$, of communication notifications.

For synchronous channels, the $\text{AFTER}\ (!, t, c.v)$ notification is matched with the corresponding $\text{BEFORE}\ (?, t', c)$ notification (i.e. matching successes with the attempts of the other other side) in order to construct the tuple $(t, c.v, t')$, which denotes that thread t communicated value v along channel c to thread t'. This tuple is then logged in the thread event model.

For buffered channels, the situation is complicated by the asynchronous behaviour. If $\text{BEFORE}\ (?, t', c)$ occurs before $\text{AFTER}\ (!, t, c.v)$, then the tuple $(t, c.v, t')$ is constructed immediately and logged in the thread event model. Otherwise, we log that $\text{AFTER}\ (!, t, c.v)$ has occurred, and should $\text{BEFORE}\ (?, t', c)$ ever occur subsequently, then the tuple $(t, c.v, t')$ will be logged in the thread event model after the occurrence of $\text{AFTER}\ (?, t', c.v)$.

For all channel types, BEFORE notifications are removed by their corresponding AFTER notifications in order to keep the model state consistent with the state of the program.

It is also necessary to notify the thread event model whenever a thread t begins to execute an alt object. This is necessary in order to log that t is attempting to perform an operation on a channel, since alt objects do not perform the channel operation themselves. The model maintains mappings from each channel c to the threads that are waiting for a shriek or query to complete on c, and maps each thread to the alt it is currently executing (if any).

2.3. Behavioural Specifications

Programmers often find it useful to use assertions within their code to help catch errors at runtime. Our tool allows the programmer to place assertions on the *patterns of communications* of the program. These assertions are constraints on event traces. *Liveness* specifications (e.g. that certain events *will* occur) are beyond the scope of the current tool.

The trait SelfLoggedEvent is a marker for objects that can be used to define behavioural specifications, known as *self-logged events*.

An instance of class Logger is used to assert trace specifications on these objects at runtime. The constructor of Logger accepts a function f: List[E] => Boolean where E can be any subtype of SelfLoggedEvent. The Logger class provides a method log(evt: E) that appends evt to the sequence of logged events, known as trace, and then asserts that f(trace) must hold. If this assertion fails, then the illegal trace is returned to the user.

As an example, consider the specification that the number of communications of a process P on a channel c always only differs by at most one from the number of communications by a process Q on channel c. This can be defined as follows:

```
abstract class CounterSpecEvent extends SelfLoggedEvent
object A extends CounterSpecEvent
object B extends CounterSpecEvent

val spec = new Logger ({
    trace: List[CounterSpecEvent] =>
      val diff = trace.count(_ == A) − trace.count(_ == B)
      0 <= diff && diff <= 1
  })

val c = ManyOne[CounterSpecEvent]
def P = proc("P"){ repeat { c!A; spec.log(A) } }
def Q = proc("Q"){ repeat { c!B; spec.log(B) } }
def Consumer = proc("Consumer"){ repeat { println(c?) } }
val System = P || Q || Consumer
```

Calling System() will run the example. In this case, it is likely that the specification will eventually fail. When it does, the illegal trace is reported to the user.

The obvious disadvantage of this scheme is that, for most specifications, the entire history of logged events must be inspected to determine whether f holds on a trace.

Therefore, behavioural specifications can also be defined as state machines. The constructor of the class StatefulLogger is parametrised with an initial state s: S, an update function u: (S,E) => S, and an assertion function g: S => Boolean, where S is the type of states of the state machine and E is any subtype of SelfLoggedEvent. The log(evt: E) method of StatefulLogger appends evt to the sequence of logged events, then transitions the state machine to the new state as defined by u and the current state, and finally checks whether the new state is legal according to g. If this test fails, then the illegal trace is returned to the user.

The above example can be reworked to use a StatefulLogger object as follows:

```
val statefulSpec = new StatefulLogger[Int,CounterSpecEvent] (
    0,
    ( diff ,evt) => evt match { case A => diff+1; case B => diff−1 },
    diff => (0 <= diff && diff <= 1)
  )
```

The process definitions are as before, except that all occurrences of spec are replaced with statefulSpec.

2.4. Deadlock Detection

In this section we describe our technique for dynamically detecting deadlock in any subnetwork of the communication graph. This relies on the construction of the contention network, which we first describe in Section 2.4.1. The algorithm proper is detailed in Section 2.4.2.

2.4.1. Constructing the Contention Network

The *contention network* is the graph with one node per thread that is waiting on some channel and an edge (s, t) if the thread of s has an ungranted request on the thread of t.

Let UNGRANTEDSHRIEKS (UNGRANTEDQUERIES) be the set of all channels c that have some thread waiting for a write (read) operation to complete on c. These sets are obtained from the attempted communications logged in the thread event model. The channels that have an ungranted request across them are exactly:

$$\text{WAITFORCHANS} = \text{UNGRANTEDSHRIEKS} \triangle \text{UNGRANTEDQUERIES}$$

where $\triangle$ is the symmetric difference operator of sets. We take the symmetric difference since any channel $c' \in \text{UNGRANTEDSHRIEKS} \cap \text{UNGRANTEDQUERIES}$ is one where both a reader and writer are attempting but neither has succeeded yet, and such occurrences are not proper ungranted requests.

Let UNGRANTEDSHRIEKEDGES (respectively UNGRANTEDQUERYEDGES) be the set of all pairs (t, c) where t is a thread that has an ungranted shriek (respectively query) operation on channel $c \in$ WAITFORCHANS, and all pairs (t', c') where t' is a thread currently executing an alt object a and c' is the channel corresponding to an output (respectively input) port used in some event of a.

These sets are constructed from the logged data: the process model logs the alt objects owned by each CSO process and the ports used in the events of those alt objects, and the thread event model logs which threads are currently running which CSO processes.

Together, UNGRANTEDSHRIEKEDGES and UNGRANTEDQUERYEDGES contain the edges (t, c) such that t is a thread that has an ungranted request across c. Note that the use of nodes representing channels is only an intermediate step, and that these nodes will not be present in the final graph.

Let INPORTEDGES (OUTPORTEDGES) contain all edges of the form (u, v) where u is a graph node representing a channel c and v is a graph node representing a process that has a reference to the input (output) port of channel c. These sets are constructed using the data logged in the process model.

It now remains to determine, for each thread t with an ungranted request, the threads that t is waiting on. We first map each thread t to the CSO processes whose executing thread may grant the request of t. Each of these processes are then mapped onto their running threads. Hence we define:

$$\begin{aligned}\text{THREADPROCEDGES} = (&(\text{UNGRANTEDSHRIEKEDGES} \circ \text{INPORTEDGES}) \\ &\cup (\text{UNGRANTEDQUERYEDGES} \circ \text{OUTPORTEDGES}))\end{aligned}$$

where $A \circ B = \{(x, z) \mid \exists y.\, (x, y) \in A \wedge (y, z) \in B\}$. The set of edges of the contention network is then just

$$E = \{(u, w) \mid \exists v.\, (u, v) \in \text{THREADPROCEDGES} \wedge w \in \text{runningThreads}\,(v)\}$$

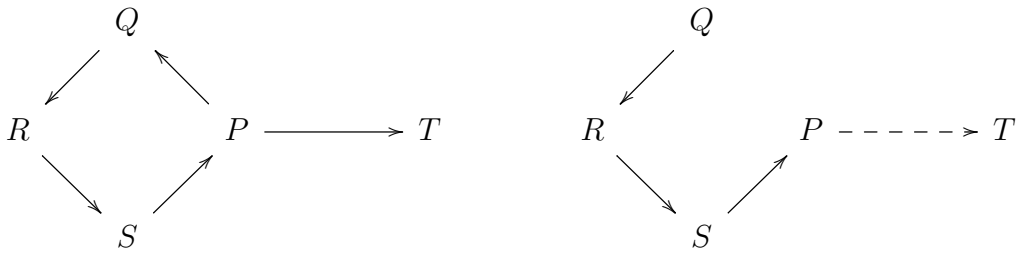
where runningThreads is the mapping from a CSO process to its executing JVM threads as maintained by the thread event model. The set of nodes (none of which represent a channel) is induced from the edge set $V = \{u, v \mid (u, v) \in E\}$. The contention network is $G = (V, E)$.

2.4.2. Deadlock Detection Algorithm

The algorithm for deadlock detection was inspired by the CTL model checking algorithm [7]. First, the contention network is obtained from the thread event model. Recall that the edges of this network are those (u, v) where u and v are threads and u has an ungranted request to v across some channel.

The existence of a cycle in the contention network does not necessarily imply that the network has entered a deadlocked state. As an example, consider Figure 7a. The contention network contains a cycle, yet this cycle is subsequently broken in Figure 7b when process P communicates with process T (shown by the dashed line) and thus no longer has an ungranted request on process Q. Therefore, the existence of a cycle is a necessary, but not sufficient, condition for deadlock [2].

A process P is deadlocked if P has an ungranted request and there does not exist a path in the contention network from P to a process Q that itself does not have an ungranted



(a) Cycle at time step t (b) Cycle broken at time step $t + 1$

Figure 7. A contention network containing a cycle (a) that subsequently resolves (b). A solid arrow from X to Y denotes that process X is waiting for (i.e. attempting to communicate with) process Y. The dashed line in (b) shows that process P communicated with process T, and is therefore no longer waiting on either T or Q.

request.[7] Intuitively, if there exists a path of ungranted requests in the contention network starting from P to another process Q that itself is not waiting on any other process, then P is not deadlocked because it is possible for that path of requests to resolve. This idea is illustrated in Figure 7.

The contention network G is a directed graph. The set of *interior nodes* is defined to be all nodes u such that there exists a node v and (u, v) is a directed edge in G. The set of *leaves* is just the complement of this set. The existence of a leaf in a directed graph is a necessary condition of acyclicity. Thus if the graph is non-empty and has no leaves then every node in the graph participates in a cycle, and the network is deadlocked.

Otherwise, if the set of leaves is non-empty, then the algorithm marks all leaves as *visited*; if a node has been visited then it is marked as deadlock free. Then, on every iteration, the algorithm marks those nodes u which have not yet been visited but for which there exists an edge (u, v) where v has been visited. The graph is declared deadlock-free if all nodes can be visited in this way. Otherwise, some sub-network is deadlocked.

The resulting algorithm is shown as Algorithm 1. It is clear that this algorithm terminates, since it inspects each node only once.

In the implementation, this algorithm is run every second in its own thread to check for deadlock, but only if the communication network contains a cycle. The user is notified within the tool if deadlock is detected.

2.5. Visualising the Behaviour of the Running Program

In this section we describe the dialogs and diagrams that provide information about the running program. The main screen of the tool is invoked by entering a question mark (?) at the console and is shown in Figure 1. In this example, the self-arrows labelled as either Sits or Pick are self-logged events. We can see that all the Philosophers have picked up the left fork and will subsequently deadlock. This deadlock is detected using the technique described in the previous section and is reported to the user.

Real-time statistics about the runtime environment are given in the top-left corner of the main screen. This data helps the user to distinguish between livelock and deadlock. The event trace displays a time-ordered list of the events communicated. The display can be paused from displaying updates, and can be set to automatically scroll to keep the most recently logged events in view.

The sequence diagram of the events communicated (shown as arrows between processes) and the self-logged events (drawn as self-arrows) is displayed on this main screen. The left-

[7] This is a sufficient, but not a necessary, condition for deadlock. For example, the network could also deadlock if a shriek or query is guarded by an **if** statement, say, and the process body enters a state where this conditional can never be satisfied in any future state of the program. Of course, determining this is an undecidable problem.

```
1: // Input: contention network G = (V, E) where
2: //   V is the set of nodes representing processes, and
3: //   E contains all edges (u, v) such that u is waiting for v
4: // Returns: true if some sub-network is deadlocked
5: function CONTENTIONNETWORK(nodes,edges)
6:     interiorNodes := {u | (u, v) ∈ edges}
7:     // The leaves are the processes not waiting for any other process
8:     leaves := nodes \ interiorNodes
9:     // If a non-empty graph contains no leaf nodes then all nodes are waiting
10:    if (nodes ≠ ∅ ∧ leaves = ∅) then return true
11:    else
12:        // Construct the adjacency list
13:        Adj := {u ↦ {v | (u, v) ∈ edges} | u ∈ nodes}
14:        // Maintain set of visited nodes (processes known not to be deadlocked)
15:        visited := leaves
16:        // Starting from the leaves, perform a backwards mark and sweep
17:        q := MAKEQUEUE(leaves)
18:        while q ≠ ∅ do
19:            next := DEQUEUE(q)
20:            for all n ∈ Adj [next] do
21:                if n ∉ visited then
22:                    visited := visited ∪ {n}   // Mark n as deadlock free
23:                    ENQUEUE(q,n)
24:        // Graph is acyclic iff we have visited all nodes
25:        return nodes = visited
```

Algorithm 1. Deadlock Detection

to-right order of the process timelines can be changed by the user, by clicking and dragging. A simple fade-back-in effect on the timelines that were switched provides a visual feedback to the user.

The concrete and abstract composition trees for the Dining Philosophers problem, as produced by the debugger, are shown in Figure 3.

The communication network constructed by the process model (see Section 2.1.1) is drawn using a force-directed algorithm. The idea is that nodes repel each other (like negatively-charged particles) but edges cause their endpoints to be attracted to each other (acting as springs). The geometry of the graph is iteratively improved from a random initial placement. Experiments indicated that this technique often gave the best output for arbitrary communication networks and regular structures (such as rings). The algorithm used is based on that presented in [8]. Experiments also showed that running the algorithm separately on each connected component of the communication graph (in the case where there are disjoint networks) produced the best results.

All diagrams produced by this tool can be exported into a variety of raster and vector file formats (including JPEG, PNG, EMF, PDF, PS, EPS, SVG, and SWF).

3. Evaluation

This section analyses the performance of our implementation using a common micro-benchmark—the Commstime test [9,10,4]. This test was used to benchmark the original CSO library implementation and thus its use is appropriate. The timing tests presented here com-

pare the original CSO and our extension of CSO with deadlock detection both enabled and disabled on a variety of machines and operating systems.

The Commstime benchmark has been used to evaluate the performance of a number of concurrency libraries [4] and is defined in Figure 8. The process network is shown in Figure 9. The Prefix process initially outputs zero which is subsequently communicated to both Succ and Consumer via an intermediate process (either SeqDelta or ParDelta). The Succ process then increments the integer it read and passes it to Prefix, which immediately outputs the value read, and this begins another round. Therefore, the cyclic network counts through all of the natural numbers sequentially, starting at zero. The Consumer process reads each integer, and records the time per cycle in microseconds averaged over batches of ten thousand reads. After each batch, the average cycle time is output.

```
val a, b, c, d = OneOne[Int]
def Prefix = proc("prefix"){ a!0; repeat { a!(b?) } }
def Succ = proc("succ"){ repeat { b!(c?+1) } }
def SeqDelta = proc("seqDelta"){ repeat { val n = a?; c!n; d!n } }
val SeqCommstime = Prefix || SeqDelta || Succ || Consumer
def ParDelta = proc("parDelta"){
  var n = a?;
  val out = proc{c!n} || proc{d!n}
  repeat { out(); n=a? }
}
val ParCommstime = Prefix || ParDelta || Succ || Consumer
```

Figure 8. Parallel and sequential Commstime networks [4].

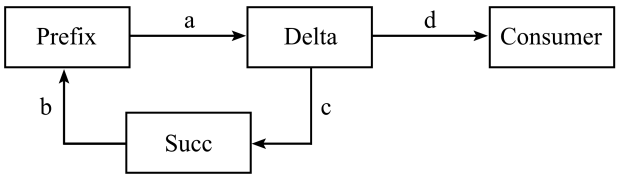


Figure 9. The Commstime network [4].

For each case, the timing tests were conducted as follows: the Commstime network was started and one thousand cycle times were recorded from Consumer. Obvious outliers (probably owing to the activity of other system processes) were removed, and then the top fifty and bottom fifty results were discarded. The greatest time and least time have been recorded as the range given in Table 1.

The sequential version of the Commstime test uses the SeqDelta that writes to Succ and Consumer sequentially, whereas the parallel version uses ParDelta that writes to these processes concurrently. Since ParDelta contains its own parallel composition of two processes, it will require a pooled thread to be unblocked and made runnable on each cycle. The overhead of this is reflected in the increased average cycle execution time for the parallel version as shown in Table 1.

The results also show that, on Linux and Solaris systems, the debugger has, on average, an overhead of less than 10% for the sequential case and closer to 2% for the parallel case when deadlock detection was disabled. Enabling deadlock detection has little effect on the sequential tests, but does increase the maxima in the range of cycle times in the parallel cases. Inspecting the raw data, the cycles that experience a slow down are those that execute while the deadlock detection algorithm is running, as we would have expected. Since the deadlock

Table 1. Commstime performance of native CSO and our extension to CSO (Logged CSO) both with and without deadlock detection (DD) enabled. Time is the average wall-time range (least to greatest) in microseconds for a single cycle of the network averaged over batches of 10000. The first column is the machine identifier. Machine 1: Intel Core 2 Duo 2×2.53 GHz, 4 GB RAM. Machine 2: AMD Athlon II X2 250 2×3.00 GHz, 4 GB RAM. Machine 3: Intel Core 2 Duo 2×2.53 GHz, 4 GB RAM. All OSs are 64-bit variants. JVM version: 1.6 update 31.

#	OS	Sequential Delta			Parallel Delta		
		CSO	Logged CSO without DD	Logged CSO with DD	CSO	Logged CSO without DD	Logged CSO with DD
1	Windows 7	14-31	17-34	20-36	120-138	132-241	135-274
	Ubuntu 10.10	25-40	27-42	30-48	54-108	58-109	59-118
	Solaris 11	25-38	26-41	29-55	58-110	57-110	61-125
2	Windows 7	12-27	18-32	19-33	124-149	145-245	145-288
	Ubuntu 10.10	24-42	26-44	28-46	57-110	58-111	58-129
	Solaris 11	24-45	28-50	28-54	59-112	59-112	60-131
3	Windows 7	15-32	18-35	21-36	121-139	138-238	140-251
	OS X 10.6	24-38	25-40	29-47	56-109	60-117	63-123

detection algorithm does not run for acyclic networks, it can be argued that this performance is acceptable for analysis purposes.

The results on Microsoft Windows, however, show greater overhead and are more varied. Interestingly, the sequential test is comparatively faster on Microsoft Windows than on either Linux or Solaris, while the parallel version is significantly slower on this operating system. This can probably be attributed to the necessary differences in the implementation of the JVM between these platforms, although one could not be sure without knowing the internals of the JVM and Microsoft Windows. While these times do not show the same performance as on Linux or Solaris, this overhead is still small considering the sophisticated models maintained.

4. Conclusion

This paper has presented a debugger for the concurrent programming library CSO. The tool maintains two models of the executing program: one describing the syntactic composition of processes and the way in which those processes may communicate, and the other logging the runtime behaviour of that system including the attempted and successful communications. The tool also allows behavioural specifications (constraints on trace patterns) to be defined in terms of self-logged events, and these specifications are checked at runtime.

Diagrams depicting these models are generated and include:

- a sequence diagram of the communications between processes and the self-logged events;
- the graph of the communication network (depicting which pairs of processes may communicate);
- the process composition trees.

This information allows the communicating behaviour of processes to be separated from the logic of the implementation of individual processes.

A particularly noteworthy feature is the dynamic detection of deadlock in any sub-network of the communication graph. Without such a mechanism, deadlock can be virtually indistinguishable from livelock.

Furthermore, the tool guarantees that any illegal usage of a concurrency primitive will be detected. This is of particular importance, since the native CSO library does not provide such guarantees and incorrect use of CSO library constructs can lead to deadlock or data loss [4].

Without such a tool, these errors can be extremely difficult to debug, since the error is not in the programmer's intended design but is in an inadvertent use of the CSO library.

Performance analysis using a common micro-benchmark provides significant empirical evidence that the overhead of the debugger is low (from approximately 2% though 10%). The importance of such a result is clear: any useful implementation of such a tool must have as little effect as possible on the timing details of the program under inspection. Of course, one should not place too much reliance on micro-benchmarks and remember that real application experience is also needed.

The ability to visualise the behaviour of the running program provides an aide-mémoire for the programmer during the construction of the program and when revisiting the code. It can also form part of the documentation of the system. These diagrams also help with debugging, particularly when the network constructed is not that was intended, for example, if the communication network unexpectedly contains a cycle. These work well on the fairly small systems one typically builds using CSO, where the number of threads is of the same order as the number of cores. However, as the number of cores on standard computers grows, so will the number of threads in systems. Visualising such large systems presents a particular challenge; we leave this for future work.

These diagrams, particularly the sequence diagram of communications and self-logged events, provide an intrinsic explanation of the extensional behaviour of the program. This information is invaluable, since conventional methods of debugging do not readily apply to concurrent programs. This is due to the interleavings of threads and because traditional debuggers report the state of all threads of the program, thus providing too much information to be useful. Our tool provides salient information relating to particular processes or sub-networks.

Finally, behavioural specifications (constraints on trace patterns) may be specified and checked at runtime in a thread-safe manner across the process network and within running threads, in a manner akin to programming with assertions but engineered for message-passing communication. Stateful behavioural specifications should scale well with the number of threads.

Although our implementation has targeted the CSO library, it should be easy to port the ideas to other libraries for message-passing concurrency, for example JCSP [11,12,13], or the Actors framework of the Scala standard library [14].

Whilst, for critical applications at least, it is often still necessary to specify the behaviour of the system in a process algebra such as CSP and perform model checking, the tool presented here can provide an invaluable insight into the behaviour of an implementation.

4.1. Related Work

Many simulators for concurrent processes have also been developed. One such example is the Ben-Ari Concurrency Interpreter, which allows the user to interactively run a concurrent program step-by-step or set breakpoints to examine the values of variables at the specified program-point [1]. However, these tools are only simulators and thus do not provide the ability to inspect a real execution, which is severely limiting considering that concurrent programs can behave very differently on different systems and configurations.

There may be many different interleavings of atomic actions, and sometimes bugs only reveal themselves after millions of iterations. Thus, stepping through code is of limited use for large examples. For such systems, a debugger that provides real-time information to the programmer at runtime may prove more useful.

An early example of an interactive debugger for message-passing concurrency is the INQUEST Transputer Network Debugger [15]. This tool allowed for the behaviour of a communicating program to be modified at runtime. Breakpoints and watchpoints could be set by

the user for specific threads. The debugger also permitted stepping through the execution of individual statements for individual threads whilst the execution of other threads in the network continued. More recently, Ritson and Simpson have developed a tool offering similar facilities for occam-π [16].

Recently, Brown and Smith have outlined how traces can be extracted from programs built using message-passing concurrency and offer proposals for how these traces could be visualised [17,18].

Deadlock detection was provided by the debugger in the very early INMOS Transputer Development System, but this required source code modification and changes to the underlying communication network [19]. A scheme similar to that presented here for deadlock detection is that of Zhang and Marwaha [20]. Their tool, known as Visputer, also produced sequence diagrams of the inter-process communications of the parallel program under inspection. These diagrams, however, were only available to the user after the network has terminated. A tool which provides real-time visualisations of the behaviour of the process network may prove more informative to the programmer.

4.2. Future Work

The implementation presented here allows for the analysis of message-passing concurrent programs. However, some very large implementations may benefit from a hybrid approach to achieving concurrency, with shared-variable concurrency localised to components which themselves communicate with message passing. Thus the work presented here could be extended to support reasoning about the behaviour and specifications of shared-variable concurrent programs, and to provide support for concurrent programs which communicate over a network. Distributed systems are an important example of the latter.

At present, when deadlock is detected, the cycle of ungranted requests is not reported to the user. It would be useful to present the programmer with the sequence of attempted communications that led to the deadlock. The thread event model of our implementation currently captures all necessary information, but it is not reported in the user interface.

Our tool does not currently support mobile channels, since it is required that all ports that a process may communicate on be listed in the process signature. However, with a little effort, it would be possible to support changing process topology by monitoring for the communication of ports across the network.

A useful extension to the existing implementation would be for historic data in the models to be cached to disk rather than kept in memory. This would allow the debugger to log an effectively unbounded number of events, since the space to log one communication is small (approximately 40 bytes), allowing the debugger to log millions of communications with realistic space requirements. Considerations would have to be given to the performance overhead of such caching.

Another extension would be to provide a mechanism to automatically clone specified objects as they are communicated or logged. (At present, only references to objects are logged.) This would mean that if an object's state was subsequently mutated then the model would still accurately reflect the communication or logging that took place. This could be achieved by providing a trait `Persistent` with a method `copy`, and simply modifying the logging method so that this method is invoked for any object of type `Persistent`.

References

[1] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, 2nd edition, 2006.

[2] A.W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[3] H. Muller and K. Walrath. Threads and Swing. *Sun Developer Network*, 2000. `http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html`.

[4] Bernard Sufrin. Communicating Scala Objects. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 35–54, September 2008.

[5] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[6] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[7] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[8] Thomas M. J. Fruchterman and Edward M. Reingold. Graph Drawing by Force-Directed Placement. *Software: Practice and Experience*, 21(11):1129–1164, November 1991.

[9] P.H. Welch. An occam Approach to Transputer Engineering. In *Proceedings of the 3rd. Conference on Hypercube Concurrent Computers and Applications*, Pasadena, California, USA, January 1988. ACM, ACM Conference Proceedings.

[10] Frederick R. M. Barnes and Peter H. Welch. Prioritised Dynamic Communicating Processes - Part I. In James S. Pascoe, Roger J. Loader, and Vaidy S. Sunderam, editors, *Communicating Process Architectures 2002*, pages 321–352, September 2002.

[11] P.H. Welch and P.D. Austin. The JCSP (CSP for Java) Home Page. *University of Kent*, 1999. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[12] Peter H. Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard H. C. Sputh. Integrating and extending jcsp. In Alistair A. McEwan, Steve A. Schneider, Wilson Ifill, and Peter H. Welch, editors, *CPA*, volume 65 of *Concurrent Systems Engineering Series*, pages 349–370. IOS Press, 2007.

[13] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Alting Barriers: Synchronisation with Choice in Java using JCSP. *Concurrency and Computation: Practice and Experience*, 22(8):1049–1062, June 2010.

[14] Philipp Haller and Martin Odersky. Actors that Unify Threads and Events. In *Proceedings of the 9th International Conference on Coordination Models and Languages*, COORDINATION'07, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag.

[15] M. Johnson. The INQUEST Transputer Network Debugger. In Jon Kerridge, editor, *Proceedings of WoTUG-16: Transputer and Occam Research: New Directions*, pages 1–10, 1993.

[16] Carl G. Ritson and Jonathan Simpson. Virtual Machine Based Debugging for occam-pi. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 293–307, September 2008.

[17] Neil C.C. Brown and Marc L. Smith. Relating and Visualising CSP, VCR and Structural Traces. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, G. S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 89–103, November 2009.

[18] Neil C.C. Brown and Marc L. Smith. Representation and Implementation of CSP and VCR Traces. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 329–345, September 2008.

[19] C. O'Neil. The TDS occam 2 debugging system. In Traian Muntean, editor, *OUG-7: Parallel Programming of Transputer Based Machines*, pages 9–14, September 1987.

[20] Kang Zhang and Gaurav Marwaha. Visputer—A Graphical Visualization Tool for Parallel Programming. *The Computer Journal*, 38(8):658–669, 1995.