Design and implementation
of a runtime deadlock detection mechanism
for SCOOP

Semester Project

Daniel Moser
danimoser@gmail.com

Supervising Professor: Prof. Dr. Bertrand Meyer
Supervising Assistant: Piotr Nienaltowski

Chair of Software Engineering
Department of Computer Science
ETH Zurich

September 2005
# Table of Contents

1 Introduction .................................................................................................................. 3

2 Background .................................................................................................................. 4

2.1 Deadlock conditions ................................................................................................. 4

2.2 Dealing with deadlocks ......................................................................................... 4

2.2.1 Deadlock prevention ....................................................................................... 4

2.2.2 Deadlock avoidance ....................................................................................... 5

2.2.3 Deadlock detection ......................................................................................... 5

2.3 Deadlock models ...................................................................................................... 8

2.3.1 Resources / communication deadlock models ............................................. 8

2.3.2 General resource system model ................................................................... 9

2.3.3 A hierarchy of deadlock models ................................................................... 9

3 Deadlock detection in SCOOP ................................................................................... 11

3.1 Design ..................................................................................................................... 11

3.1.1 A simple approach ....................................................................................... 11

3.1.2 Introducing synchronous locks ..................................................................... 12

3.1.3 Adding asynchronous locks ......................................................................... 12

3.1.4 Adding support for lock passing .................................................................... 15

3.2 Implementation ....................................................................................................... 19

3.2.1 Class overview ............................................................................................. 19

3.2.2 New classes ................................................................................................... 20

3.2.3 Modified classes ........................................................................................... 22

3.3 Limitations ................................................................................................................ 26

3.3.1 Unrecognizable deadlocks .......................................................................... 26

3.3.2 Distributed scheduling .................................................................................. 27

4 Benchmarks ................................................................................................................ 28

4.1 Producer consumer ............................................................................................... 28

4.2 Reader and modifier .............................................................................................. 29

4.3 Incrementing .......................................................................................................... 30

4.4 Evaluation ............................................................................................................... 30

5 References .................................................................................................................. 32
1 Introduction

Deadlocks are one of the major problems in concurrent programming. A deadlock happens when some tasks are blocked forever because their requests for resources will never be satisfied. In order to preserve concurrency, we must get deadlocks under control.

This report is organized as follows. Section 2 discusses the problem of deadlocks. First, we list the necessary and sufficient conditions for deadlock. We also discuss the three popular ways of dealing with the problem; we analyze their applicability to SCOOP. At the end of that section, different deadlock models are introduced and compared.

In section 3, we describe the design of a deadlock detection algorithm for SCOOP. We begin with a simple and general model for deadlock detection and then refine and improve the algorithm step by step until all desired features of SCOOP (including the lock passing) are supported. In section 3.2 we provide all implementation-related information that is necessary for future extensions of the mechanism. The limitations of deadlock detection, and specifically of the algorithm that was implemented, are discussed in section 3.3.

Two very important requirements for deadlock detection are correctness and good performance. The two conditions for correctness are that all existing deadlocks are detected (no underreporting) and that no false deadlocks are detected (no overreporting). These issues are also discussed in section 3.

Although correctness has the highest priority, we want to perform deadlock detection as efficiently as possible. We set an upper limit on the acceptable overhead induced by the deadlock detection mechanism at 50%. In order to assess the performance of the algorithm, we need to run benchmarks and compare the overhead that is created when using deadlock detection on a non-deadlocking application to running the same application without deadlock detection. Also we distinguish between the executions with assertions (preconditions) turned on and without assertions.
2 Background

2.1 Deadlock conditions

Deadlocks can only arise if all of the conditions below are satisfied:

- **Mutual exclusion:**
  Only one processor can access a resource simultaneously.

- **Hold and wait:**
  All previously acquired resources are held while waiting for other resources.

- **No preemption:**
  A resource cannot be taken from a processor which acquired it, unless the processor itself releases it.

- **Circular wait:**
  All processors involved in a deadlock form a cycle in a graph where the processors are nodes and the requested locks are directed edges.

If at least one of these conditions is not satisfied, then there cannot be any deadlock.

2.2 Dealing with deadlocks

There exist three ways of dealing with deadlocks: *deadlock prevention*, *deadlock avoidance* and *deadlock detection* [Shi90].

2.2.1 Deadlock prevention

As described in section 2.1, there are four conditions that have to be satisfied simultaneously in order for a deadlock to happen. The goal of *deadlock prevention* is to ensure that, at any time, at least one of these conditions does not hold.

Usually, this goal can be achieved in two ways:

1. Acquire all locks in advance. If this is not possible, release all the locks acquired so far. The *hold and wait* condition does not hold in that case.

2. Allow processor P₂ with a higher priority to preempt processor P₁ that holds the resource required by P₂. In this case, the *no preemption* condition does not hold.
3. A processor may only acquire the requested locks in a strictly ascending (or descending) order. That order is statically defined, e.g. all resources are numbered. By doing so, the *circular wait* condition is avoided.

### 2.2.2 Deadlock avoidance

Deadlock avoidance is based on the use of extra information by the runtime system to predict and avoid deadlock situations.

Nevertheless, especially for distributed systems, this approach is usually very inefficient:

- The system concurrency becomes restricted because some tasks cannot be executed simultaneously to avoid deadlocks. And deciding whether a deadlock can occur or not is computationally expensive, especially in distributed systems where there are many resources and tasks.
- Because there is no global synchronization mechanism in distributed systems, tasks might be required to acquire locks in advance.

In many systems, future resource requests are unpredictable, and therefore deadlock avoidance is impossible.

### 2.2.3 Deadlock detection

Instead of detecting and resolving deadlocks in advance, we let the system run into deadlocks, detect deadlocked processors and resolve the deadlocks at runtime. The deadlock detection is performed in parallel with the program execution.

The goal is to maintain a graph structure during the execution and check this graph for cyclic waits. Depending on the system we are dealing with, the authors of [Shi90] suggest one of the following graph types:

**Task Wait-For Graph (TWFG):**

The simplest possible graph for deadlock detection is the Task Wait-For Graph. The only nodes inside these graphs are tasks because in this scenario the tasks are also resources at the same time, hence there is no need for additional nodes for resources. The edges represent the wait relation between two tasks.
In the context of databases, the tasks usually are transactions which perform a series of database operations. Therefore a TWFG is also called a Transaction Wait-For Graph.

In the example below, task T2 waits for task T1, T1 waits for T3, T3 waits for T4 and T4 waits again for T1. The tasks T1, T3 and T4 create a cycle and therefore cause a deadlock. Because T2 also waits for a task involved in a cycle (T1), it is also deadlocked.

Task-Resource Graph (TRG):
A more sophisticated graph is the Task-Resource Graph, which is a bipartite directed graph. The two subsets of vertices are the tasks \( T = \{T_1, T_2 \ldots T_m\} \) and the resources \( R = \{R_1, R_2 \ldots R_n\} \).

There are two different semantics for the edges in a TRG. An edge is either a resource request or a resource assignment. Request edges \((T_i, R_j)\) always point from a task \( T_i \) to a resource \( R_j \), whereas assignment edges \((R_k, T_l)\) are directed from a resource \( R_k \) to a task \( T_l \), which is the holder.

In the example below, the resource \( R_1 \) is locked by \( T_1 \), \( T_1 \) requests \( R_2 \) which is locked by \( T_2 \) and \( T_2 \) requests \( R_1 \). This sample graph forms a cycle, so the tasks \( T_1 \) and \( T_2 \) are deadlocked.

\[ T_1 \xrightarrow{\text{Lock or Request}} T_2 \]
\[ \xrightarrow{} T_3 \]
\[ \xrightarrow{} T_4 \]

\[ \xrightarrow{} T_1 \]

Figure 1: A Task Wait-For Graph

\[ A \quad \text{bipartite graph’s vertices are partitioned in two disjoint subsets such that there are only edges from vertices of one subset to vertices of the other subset.} \]
General Resource Graph (GRG):

A generalization of the TRG is the General Resource Graph which is used to describe Holt’s General Resource Systems [Hol72]. The resource vertices are split up into two disjoint subsets – the reusable resource subset $RR = \{RR_1, RR_2 \ldots RR_x\}$ and the consumable resource subset $CR = \{CR_1, CR_2 \ldots CR_y\}$.

Each reusable resource $RR_i$ has a total number of units and for each consumable resource $CR_j$ there is at least one task $T_k$ which is the producer of $CR_j$.

In a General Resource Graph, there are three types of edges:

- Request edge:
  
  A request edge $(T_i, RR_j)$ or $(T_i, CR_k)$ from a requesting task $T_i$ to the requested resource $RR_j$ or $CR_k$ and represents a pending request.

- Assignment edge:
  
  An assignment edge $(RR_i, T_j)$ is an edge directed from a reusable resource $RR_i$ to an assigned holder $T_j$. The total number of assignments directed from $RR_i$ can never be larger than the total number of units of $RR_i$.

- Producer edge:
  
  A producer edge $(CR_i, T_j)$ exists if and only if $T_j$ produces $CR_i$.

In the example below, Task $T_1$ holds two out of five units of the reusable resource $RR_1$ and requests one unit of the consumable resource $CR_2$. Task $T_2$ is the only producer of $CR_2$ and also holds one unit of $RR_1$. Assignment and Request edges may disappear with state changes and are therefore displayed as solid, whereas the producer edges are permanent and are drawn as dashed arrows.
2.3 Deadlock models

In this section we give a short overview of the existing deadlock models.

2.3.1 Resources / communication deadlock models

In this model, deadlocks are divided into two different types – *resource deadlocks* and *communication deadlocks*.

**Resource deadlocks**

In resource deadlocks, tasks access resources (e.g. threads access shared memory), and the resources have to be acquired before they are used. If a task cannot lock the desired resource, then it has to wait until the resource was freed by the task which currently has the lock on the resource.

Trying to match this scenario to a graph type, we realize that we have two kinds of nodes – resources and tasks. The locks are represented by directed edges. Therefore, resource deadlocks can be detected using a *Task-Resource* Graph (TRG): if tasks hold the resources and therefore the tasks themselves have to be locked, we end up with a graph which has only nodes of type task, which corresponds to a *Task Wait-For Graph* (TWFG): this is the scenario used for SCOOP.

Deadlock detection with resource deadlocks corresponds to cycle detection in the TWFG.

**Communication Deadlocks**

In communication deadlocks, the resources that tasks are waiting for are messages (or synchronization between tasks), and not shared memory. Also a cycle detection algorithm is not suitable for detecting deadlocks in this scenario; more advanced algo-
algorithms are necessary. This kind of deadlock is irrelevant for SCOOP, given the assumption of the centralized execution platform provided by SCOOPLI.

### 2.3.2 General resource system model

The general resource system model was introduced by R. C. Holt [Hol72]. It merges resource deadlocks with communication deadlocks. This model uses a *General Resource Graph* (GRG):

For SCOOP, this model is too complex because it includes also communication deadlocks; therefore it is not discussed any further.

### 2.3.3 A hierarchy of deadlock models

Edgar Knapp [Kna87] proposed a hierarchical set of deadlock models to describe characteristics of deadlocks in a system. The models proposed by him range from very restricted models of request forms to completely unrestricted models.

In the following sections, the most important models will be discussed.

**Single-resource model**

The simplest possible model is the single-resource model. Each task can only have one outstanding request and therefore, in a directed graph, only one outgoing edge.

In this scenario, a *Task-Resource Graph* (TRG): is sufficient and deadlocks can be detected if there is a cycle in that graph. All deadlocked tasks can be obtained by getting all tasks in a cycle and by adding all tasks that can only reach deadlocked tasks.

For SCOOP, this scenario is too simple and does not meet all the requirements - SCOOP allows a processor to request more than one resource at a time.

**AND model**

The AND model allows tasks to request a set of resources and the tasks are blocked until they have acquired all requested resources. So each task can have multiple outgoing edges.

As in the single-resource model, it is still sufficient to build up a *Task-Resource Graph* (TRG): and detect cycles in that graph.
This is the appropriate model for SCOOP. Nevertheless, because processors represent both the tasks and the resources in SCOOP, a simpler Task Wait-For Graph (TWFG): is sufficient.

**OR model**

In the OR model, tasks are blocked only until one of the requested resources has been acquired, in contrast to the AND model. This model is useful for example in databases which have data replication and a task only needs read access to one copy of the desired resource.

In contrast to the AND model, detecting a cycle in a General Resource Graph (GRG): is not a sufficient condition for detecting deadlocks anymore. Instead, detecting deadlocks in the OR model can be reduced to finding a knot\(^2\) in the GRG. Therefore a task T is deadlocked if it is in a knot or T can only reach deadlocked tasks.

This model is not suitable for SCOOP because processors always have to lock all requested resources to continue.

---

\(^2\) A knot K is a nonempty set of vertices such that the reachable set of each vertex v in K is exactly knot K.
3 Deadlock detection in SCOOP

In the first part of this section we present the design of the deadlock detection mechanism. Starting with a basic approach for deadlock detection, we will extend the design more and more by adding support for asynchronous locks and lock passing in order to fulfill the requirements of the current SCOOP implementation.

We point out our contributions to SCOOP in the implementation section. We also discuss the limitations of the implementation.

3.1 Design

SCOOP processors correspond to tasks as introduced in the discussion in section 2.2. But these processors represent not only tasks but also resources - processors always lock the processor handling the requested resource (and not the resource itself) in order to get exclusive access.

Because the only nodes in the deadlock detection graphs are processors, a Task Wait-For Graph (TWFG): is sufficient for our purposes.

3.1.1 A simple approach

We already determined the nature of nodes in the deadlock detection graph. We have not yet reasoned about the types of edges that we need.

Basically, there are two kinds of edges:

- Wait edges:

  A processor $P_1$ requests another processor $P_2$, but $P_2$ is already locked by another processor $P_3$. So $P_1$ has to wait until $P_3$ releases $P_2$.

- Lock edges:

  A processor $P_3$ has locked processor $P_2$, and as long as $P_3$ holds the lock on $P_2$, the edge remains in the graph.
With this scenario two different deadlock scenarios can occur:

1. A processor P₂ which is locked by processor P₁ tries to get a lock on P₁.
2. Two processors P₁ and P₂ which are locked by some other processors try to lock each other.

In both scenarios, the deadlock can be detected because the edges form a cycle.

3.1.2 Introducing synchronous locks

When taking a closer look at the two scenarios in the previous section (Figures 4 and 5), we realize that wait edges and lock edges share the same properties – a processor P₁ gets blocked until its request has been processed and the locks have been released. The only difference is that, for a wait edge, P₁ is blocked longer because it first has to acquire the lock before processing the request. But this does not influence deadlock detection.

For deadlock detection we can therefore treat wait and lock edges as one edge type: synchronous edges.

3.1.3 Adding asynchronous locks

In the previous two sections we always assumed that if a processor P₁ gets the lock on another processor P₂, it will wait until its request has been finished. But in SCOOP, not all calls which cause locking are synchronous. In fact, a call only has to be synchronous if the calling processor P₁ is waiting for a result from its feature call to processor P₂, or if lock passing occurs (see next section).
If an algorithm detects a deadlock which is in fact no deadlock, we call this a **false deadlock**. In our algorithm a false deadlock can be detected when considering only synchronous edges, as shown in the following example:

```ruby
class A
  feature
    r (x: separate B) is
      do
        x.foo
      end
    callback is
      do
        -- Do something
      end
end

class B
  feature
    caller: separate A
    -- Caller of foo
    foo is
      do
        callback (caller)
      end
    callback (x: separate A) is
      do
        x.callback
      end
end
```

What happens is that an object \(o_A\) of type A on processor \(P_1\) gets a lock on an object \(o_B\) of type B on processor \(P_2\), and \(o_B\) then tries to get a lock on its caller \(o_A\). But because \(o_A\) only calls \(o_B\) asynchronously by calling \(x.foo\), \(o_A\) will eventually become unlocked and then it is possible for \(o_B\) to get a lock on \(o_A\).

But when treating all calls as synchronous, the following situation arises:

![False deadlock diagram](image)

**Figure 6: False deadlock**

There is a cycle and we report a deadlock. But what we cannot express is that the edge \((P_1, P_2)\) will soon disappear because it was an asynchronous call, and as soon as \(P_1\) becomes unlocked the situation in the deadlock graph looks like this:
To avoid false deadlock reporting, we have to add a new edge type to our deadlock detection graph. We call it **asynchronous edges**.

Since we know that the asynchronous edge will sooner or later disappear (or turn into a synchronous edge when a synchronous call is made), we do not include it into our cycle detection algorithm. The example used before looks better now:

![Figure 8: No false deadlocks with asynchronous edges](image)

Because the asynchronous edge \((P_1, P_2)\) is not included in the cycle detection, no deadlock will be detected anymore. **Note:** The edge \((P_2, P_1)\) is still synchronous because \(P_2\) is waiting for a lock on \(P_1\).

We solved the problem of false deadlocks. But there is still another problem: there are deadlocks that are not recognized by the algorithm.

In the following scenario, processor \(P_1\) has locked processor \(P_2\) synchronously and processor \(P_3\) asynchronously. During execution, \(P_2\) waits for a lock on \(P_3\), but because \(P_1\) will not release \(P_3\) as long as it is blocked waiting for a result from \(P_2\), this scenario creates a deadlock, although there is no cycle in the graph.

![Figure 9: Undetected deadlock](image)
This problem only arises when processor P_1 has a synchronous lock and some asynchronous locks to other processors at the same time.

The solution to this problem is actually not difficult – we only need to modify the cycle detection algorithm in a way that if there is a synchronous outgoing edge from P_1 then it considers asynchronous edges from P_1 to other processors as synchronous edges from those other processors to P_1. Then a cycle will be formed and the deadlock will be detected.

### 3.1.4 Adding support for lock passing

Since the SCOOP model has been enriched with the support for lock passing, we need to cater for that mechanism too. The necessary extension to the deadlock detection algorithm is relatively small but non-trivial. In fact, the lock passing mechanism allows for much more sophisticated synchronization patterns than the original SCOOP model; this results in a much richer repertoire of possible deadlock patterns.

The following scenario uses the lock passing mechanism:

```plaintext
class A
feature
  r (x: separate B; y: separate C) is
    do
      x.foo (y)
    end
end
class B
feature
  d: separate D
  foo (x: separate C) is
    do
      bar (d)
      -- Do something with x
    end
  bar (y: separate D) is
    do
      -- Do something
    end
end
```

When the feature r of object o_A of type A on processor P_1 is called, P_1 will obtain a lock on processor P_2 and P_3 which hold the objects o_B of type B and o_C of type C respectively. In the deadlock detection graph, P_1 has asynchronous locks on P_2 and P_3.
In the next step, the lock passing occurs when \( x.\text{foo}(y) \) is called on object \( o_B \). The asynchronous edge (\( P_1, P_2 \)) has to be changed to a new type of edge; we call it **lock passing edge**. When receiving the locks, \( P_2 \) has the locks for \( P_1 \) and \( P_3 \) (dash-dotted lines) and when calling \( \text{bar}(d) \), \( P_2 \) locks another Processor \( P_4 \) which holds an object \( o_D \) of type \( D \) asynchronously.

The current situation in the graph can be visualized as shown in Figure 11:

![Diagram of lock passing](image)

**Figure 10:** Before lock passing

**Figure 11:** Lock passing from \( P_1 \) to \( P_2 \)

**Changing lock types**

In Figure 11: **Lock passing from \( P_1 \) to \( P_2 \)** the lock that was passed was asynchronous. A passed lock can never be synchronous because each node can only have one outgoing synchronous edge, and this is the edge that is used for lock passing.

But once \( P_2 \) has received the lock, it may change the lock to synchronous when executing another feature.
Figure 12 shows the deadlock detection graph after P2 has changed the lock on P3 to synchronous. It is important that the original edge from P1 to P3 is changed to synchronous and no new edge is inserted between P2 and P3, although P2 is changing the lock. Otherwise we would have an asynchronous edge from P1 to P3 and a synchronous edge from P2 to P3 which would cause a false deadlock.

In Figure 13 the resulting false deadlock is illustrated. Because the asynchronous edge is turned in the opposite direction for deadlock detection, a cycle would appear between P1, P2, and P3 (we still consider the lock passing edge as synchronous).

**Merging Processors**

We can reduce the number of nodes that appear in the cycle detection by merging all processors in a lock passing chain into one processor. All the edges between the merged processors can be neglected for the cycle detection purposes but all the edges leading from a merged processor to another merged or non-merged processor have to be considered.
Figure 14 represents the graph from Figure 11 as it is seen by the deadlock detection algorithm. The processors P_1 and P_2 are merged into one processor P_{1/2}; all edges between P_1 and P_2 have disappeared but all other edges are still visible.

Having merged all processors, we can run the deadlock detection algorithm to detect deadlocks. The advantage of merging the nodes is that the cycle detection performs better because there are fewer nodes and edges to consider.
3.2 Implementation

3.2.1 Class overview

Figure 15 shows SCOOPLI classes that implement the deadlock detection algorithm.

Three new classes were introduced to perform the deadlock detection. The main class is `SCOOP_DEADLOCK_DETECTOR`. `SCOOP_DEADLOCK_NODE` and `SCOOP_DEADLOCK_EDGE` are classes which implement the graph representation.

Major changes were made to the existing SCOOPLI in `SCOOP_SCHEDULER`, minor changes in `SCOOP_PROCESSOR`, `SCOOP_ROUTINE_REQUEST` and `SCOOP_SEPARATE_PROXY`. 

Figure 15: Important classes for deadlock detection
3.2.2 New classes

**SCOOP_DEADLOCK_DETECTOR**

This class maintains the graph representation of the processors and the locks and also controls the synchronization of the graph. It has functionality to add and remove both the processors and the locks, to perform deadlock detection and also to report deadlocked processors.

Here is a list of the most important features of this class:

- **add_processor (a_proc: SCOOP_PROCESSOR)**
  
  Add processor a_proc to the graph.

- **remove_processor (a_proc: SCOOP_PROCESSOR)**
  
  Remove processor a_proc from the graph.

  For performance reasons, remove_processor does not check whether all edges have been removed previously or remove remaining edges; it simply deletes the processors. If the program executes correctly, there should not be any edges left when remove_processor is called. We considered adding a corresponding precondition but it resulted in a poor performance due to the nature of such precondition (graph traversal).

- **set_sync_lock (a_node_1, a_node_2: SCOOP_PROCESSOR)**
  
  Check if there exists an edge between a_node_1 and a_node_2 and set the edge type to synchronous (no matter what the original type was); if the edge does not exist, a new edge will be created and set to synchronous.

- **set_async_lock (a_node_1, a_node_2: SCOOP_PROCESSOR)**
  
  Similar to set_sync_lock but for asynchronous edges.

- **remove_lock (a_node_1, a_node_2: SCOOP_PROCESSOR)**
  
  Remove an edge from the graph. This is only done when the execution is finished and the locks are released. In case the locks could not be acquired, the edge is not removed but set to synchronous to wait (see section 3.1.2).

- **has_deadlock: BOOLEAN**
This feature implements the cycle detection algorithm discussed in section 3.1.

The cycle detection is implemented using a topological sorter which is part of the extension of EiffelBase written by Olivier Jeger as a Master thesis [Jeg04]. In the topological sorter, we insert all processors which are not involved in lock passing; for all lock passing chains, we insert the merged node. In the implementation, we simply consider the node which initiated the lock passing to be the merged node, so we do not have to create a now node. See also section 3.1.4.

As constraints for the topological sortser, we insert all the necessary edges that were discussed in section 3.1.

After inserting all the nodes and constraints, we let the topological sorter process the graph. If there is a cycle (and therefore the topological sorting is not possible), the feature \textit{cycle_found} of the topological sorter will yield \textit{true} and all deadlocked processors will be marked as such by calling \textit{mark_processor_as_deadlocked}.

- \textbf{deadlocked_processors: LINKED_LIST [SCOOP_PROCESSOR]}

  This feature iterates over all processors in the deadlock detection graph and returns all processors that are marked as deadlocked.

- \textbf{processors: LINKED_LIST [SCOOP_DEADLOCK_NODE]}

  This list contains all objects of type \textit{SCOOP_DEADLOCK_NODE}. This is the representation of our deadlock detection graph. The edges of the graph are accessed via the nodes.

- \textbf{mark_processor_as_deadlocked (a_node: SCOOP_DEADLOCK_NODE)}

  When a processor is deadlocked, then all processors which are synchronously locked by that processor are also deadlocked (the processors do not necessarily have to be in a cycle). Therefore this feature marks as deadlocked all processors that have synchronous locks to a deadlocked processor or that are synchronously locked by a deadlocked processor.

\textbf{SCOOP_DEADLOCK_NODE and SCOOP_DEADLOCK_EDGE}

The classes \textit{SCOOP_DEADLOCK_NODE} and \textit{SCOOP_DEADLOCK_EDGE} are used for representing the deadlock detection graph.
Each node has a list of outgoing and incoming edges which will be automatically updated when adding or removing edges. Additionally, a deadlocked node can be marked as deadlocked, so it is not considered in the next deadlock detection.

Edges have a source and a target node as well as a label that represents the different types of edges.

As an alternative, the Graph Library which is also part of the extension to EiffelBase by [Jeg04] could be used to represent the graph. Initially, we based our solution on that library but it turned out to be less efficient than required. The library is very powerful and provides many general mechanisms but this comes at a price of lower efficiency.

### 3.2.3 Modified classes

**SCOOP_SCHEDULER**

Most of the modifications required for the deadlock detection mechanism were done in **SCOOP_SCHEDULER**. Besides calls to **SCOOP_DEADLOCK_DETECTOR**, the following implementation related features were added:

- **adjust_max_attempts**

  It would be a bad idea to initiate the deadlock detection cycle every time a lock could not be acquired, as the lock might already be acquired in the next attempt. Therefore the performance would be affected seriously when running the deadlock detector too often.

  We need some reasonable way of deciding when the deadlock detection should be initiated. One option is to starts deadlock detection after \( x \) unsuccessful locking attempts (\( x \) being constant). But then a problem arises for some applications that tend to have many deadlocks - it takes too long until the deadlock detection starts. Conversely, for other applications which never have deadlocks, there would be still too many detection cycles. Therefore it is better to vary the maximum number of unsuccessful locking attempts. The feature **adjust_max_attempts** compares the number of successful deadlock detections to the total number of deadlock detections and adjusts **max_attempts** according to the following scheme:
<table>
<thead>
<tr>
<th>Success Rate (X)</th>
<th>Action on max_attempts</th>
</tr>
</thead>
<tbody>
<tr>
<td>X ≤ 25 %</td>
<td>max_attempts := max_attempts * 2</td>
</tr>
<tr>
<td>25 % &lt; X &lt; 75 %</td>
<td>max_attempts remains unchanged</td>
</tr>
<tr>
<td>X ≥ 75 %</td>
<td>max_attempts := max_attempts // 2</td>
</tr>
</tbody>
</table>

- **deadlock_interval_reached: BOOLEAN**

  The programmer can set a time limit in milliseconds to prevent deadlock detection to be executed too often by calling the appropriate feature inherited from `SCOOP_CONCURRENCY`. The feature `deadlock_interval_reached` determines whether the last deadlock detection was already executed a certain number of milliseconds ago. A deadlock detection cycle will be initiated only if both conditions, the maximum number of attempts and the deadlock interval, have been satisfied.

- **execute_thread**

  This feature takes the next `routine_request` and tries to acquire the all the locks required for execution. If it does not succeed to acquire all the locks, the number of unsuccessful attempts is incremented. As a result, if the two conditions discussed before are satisfied, a deadlock detection cycle is initiated and existing deadlocks, if any, are reported.

  After the number of deadlock detections is larger than the maximum number of detections `max_deadlock_detections`, the feature `adjust_max_attempts` is called to adapt the maximum number of unsuccessful locking attempts `max_attempts`. 
• **locks_acquired (...): BOOLEAN**

This feature is very important for the deadlock detection algorithm it implements the edge insertion. The inserted edges represent either the lock that were acquired (asynchronous edges) or the locks that were not acquired and therefore remain requests (synchronous edges).

In SCOOP, locks are acquire atomically, that is either all or no locks can be acquired; if at least one of the locks cannot be acquired, then all other locks will be released.

This means that if all locks could be acquired, then all edges that are inserted in the graph are asynchronous. Otherwise, all lock requests have to be inserted as synchronous edges the first time when locking fails. If locking fails again in a second attempt, then the edges are already synchronous and do not need to be updated.

**SCOOP_PROCESSOR**

• **proc_id: INTEGER**
Each processor has a unique id which is required for the deadlock detection graph. It is assumed that there are never more different processors than can be represented by integers.

- **synchronous_processors_top: SCOOP_PROCESSOR**

  Return processor $P_i$ at the top of the stack of synchronous processors. If there is no lock passing, $P_i$ corresponds to the current processor. In case of lock passing, $P_i$ corresponds to the processor which initially started lock passing.

  For lock passing, we need to merge all the nodes involved in a lock passing chain. Instead of creating a new node, we simply use the “root node” of the lock passing chain. Therefore $P_i$ is chosen to represent the merged node.

**SCOOP_ROUTINE_REQUEST**

There should be a variable number of attempts to acquire the locks for each routine request before the deadlock detection starts. For that reason, **SCOOP_ROUTINE_REQUEST** was extended by a counter and some features to increment and reset the counter. The counter is used by the **SCOOP_SCHEDULER**.

**SCOOP_SEPARATE_PROXY**

When a lock is acquired by a processor $P$, an asynchronous edge is added to the deadlock detection graph by default. But as soon as $P$ calls a routine on the locked resource and $P$ is waiting for the result of the call or has its locks passed, the edge must become synchronous for the duration of the call; afterwards it becomes asynchronous again.

In case of lock passing, no locking takes place, therefore we do not need to create and change any edges. In order to recognize lock passing we check if there is an (asynchronous) edge between the involved processors in the graph. If there is no edge, we are dealing with lock passing; otherwise we convert the asynchronous edge into a synchronous one.

- **scoop_synchronous_execute (...)**
3.3 Limitations

3.3.1 Unrecognizable deadlocks

In SCOOP, there are three different kinds of deadlocks which can occur:

- **Resource Deadlocks**
  
  Two or more processors cannot proceed with their execution because they request locks from one another.

- **Infinite Loops**

- **Wait-conditions that are not satisfiable**
  
  Although all requested locks are available, the feature cannot be executed because its wait-condition is never going to be satisfied.

*Resource deadlocks* can be detected by the algorithm introduced above, but there is no possibility to determine at runtime if a loop ever terminates or if a wait-condition can ever be satisfied, so *infinite loops* and *unsatisfiable wait-conditions* are not detected by this algorithm. But to prevent *infinite loops*, other mechanisms exist in SCOOP, namely *wait conditions* and *invariants*.
3.3.2 Distributed scheduling

The implementation of this deadlock detection algorithm is a centralized solution. This means that there is a centralized scheduler that controls all the locking and also cares about deadlock detection.

The presented approach cannot be applied as is in a distributed context. The lack of a centralized scheduler would violate the simplifying assumptions we made. Communication delays and failures would need to be reflected. As a result, more complex deadlock detection schemes would be necessary to detect deadlocks.

Nevertheless, our work constitutes a solid basis for the development of a distributed deadlock detection algorithm. In particular, our approach to dealing with the problem of lock passing might be a rich source of inspiration.
4 Benchmarks

To evaluate the performance of the implementation, we created different test applications to cover various scenarios. In order to compare it to an implementation of SCOOP without deadlock detection, our test applications must not cause deadlocks and therefore need to be correct. We also should avoid using a log-file or console output because that might cause undesired overhead and falsify our benchmarks considerably.

In all three test cases, the benchmarks were done once with precondition checking turned on and once without precondition checking. All benchmark applications have been finalized and executed without EiffelStudio.

The test system is an AMD Athlon64 3700+ (San Diego) with 1 MB of L2 Cache, 1024 MB of memory and is running Windows XP Professional. The tests were performed in Safe Mode to prevent other processes from influencing the benchmarks.

The SCOOP code was preprocessed with scoop2scoopli 0.2.2001 and compiled using EiffelStudio 5.5. We used SCOOPLI 0.3.2001.

4.1 Producer-consumer

In this scenario, a bounded buffer of size $l$ represents a shared resource. There are $m$ producers that generate $i$ new values and insert them into the buffer, and $n$ consumers that remove $j$ values from the buffer one by one. Every producer and consumer needs an exclusive access to the buffer in order to modify it.

Additionally, two wait conditions exist: producers may only insert values as long as the buffer is not full, and consumers may only consume when the buffer is non-empty.

The length $l$ of the buffer is set to 5; the other parameters are varying in the benchmarks.
Interesting in the result of this benchmark is that increasing the number of productions and consumptions does not increase the overhead for deadlock detection significantly. But when increasing the total number of producers and consumers, the overhead becomes slightly bigger because there are more processors in the deadlock detection graph and hence insertion and deletion of edges between processors takes more time.

### 4.2 Reader and modifier

This benchmark tests how efficient lock passing performs in combination with normal locking. There is a buffer $b$ that stores an integer, a modifier $m$ and $k$ readers. The readers try to get exclusive access to the buffer. Once the lock has been acquired, the value in the buffer is read; then the lock is passed to the modifier, which increments the value stored in the buffer before the lock is returned to the reader and released. This process is repeated $n$ times.
This benchmark confirms what we already observed in 4.1: The number of repetitions does not have a considerable influence on the overhead created by deadlock detection.

### 4.3 Incrementing

The last benchmark determines the overhead incurred by the repeated (multi-stage) lock passing. In the benchmark, a shared buffer is passed on 10 times from one instance of `INCREMENTER` to another instance; the value stored in the buffer is incremented at each step. When the value reaches 10, it is reset to 0. This process is repeated \( n \) times, so we totally have \( 10 \times n \) lock passing operations.

Because there are only four instances of `INCREMENTER`, the lock passing chain is cyclic.

![Graph showing overhead for different conditions and values of \( n \).]

The overhead for deadlock detection is smaller than in the first two benchmarks. One of the reasons is that the original SCOOP has to perform several operations for each lock passing, e.g. pushing stacks on top of other stacks and locking the target processor. But for deadlock detection there is no additional overhead because it only has to add one synchronous edge whenever a lock passing occurs.

Again, we observe that the number of repetitions does not affect the overhead.

### 4.4 Evaluation

In section 1 we aimed at reaching an overhead of 50% or less compared to the original SCOOP/LI.
Although in some cases with precondition checking the overhead is up to 45%, the more important benchmarks where precondition checking was turned off scored all below 30%. We can assume that, in real-life examples, not all locking requests correspond to lock passing. Therefore, an average overhead will be around 25 to 30%, which we consider to be satisfactory.

But we also have to consider that our benchmarks were designed not to cause any deadlocks (otherwise we would not be able to compare the results with the original SCOOP). Hence many parts of the implementation were not touched by the benchmarks and cannot be evaluated until there is an alternative implementation of deadlock detection or other implementations which serve as comparisons (e.g. deadlock prevention or deadlock avoidance mechanisms).
5 References


