A Transformation to Provide Deadlock–Free Programs*

Pablo Boronat and Vicente Cholvi

Dept. LSI, Universidad Jaume I, Campus Riu Sec s/n,
12071Castellón, SPAIN
{boronat, vcholvi}@lsi.uji.es

Abstract. A commonly employed technique to control access to shared resources in concurrent programs consists of using critical sections. Unfortunately, it is well known that programs using several critical sections may suffer from deadlocks.
In this paper we introduce a new approach for ensuring deadlock–freedom in a transparent manner from the programmer's point of view. Such an approach consists of obtaining a deadlock–free "version" of the original program by using some program transformations. We formally prove the correctness of those transformations and we analyze their applicability.

1 Introduction

In the development of concurrent programs, considerable effort has been devoted to study the resource allocation problem. While local resources are accessed only by one process, shared resources can be accessed by many. If two or more processes simultaneously use the same resource, they can leave it in an incoherent state. Therefore, some mechanism is necessary to ensure that at most one process is accessing a shared resource at a time.

Maybe the most widely used mechanism for such a task consists of using *critical sections*. Roughly speaking, a critical section can be seen as a section of code where processes have exclusive access to the resources allocated within.

In order to access a critical section, a process must acquire and release it by executing especial code. This code forces shared resources to be "sequentially" accessed. Consequently shared resources are left in coherent states. Unfortunately, it is widely-known that programs which use several critical sections may suffer from deadlocks (i.e., the program stands in an infinite wait).

In order to avoid deadlocks, three approaches have been traditionally used. The first approach consists of detecting when a deadlock occurs and then recovering to a safe state (i.e., a previous state where the program is not deadlocked) [10,7]. The second approach prevents deadlock states in execution time using knowledge about the current state [3], in particular the state of the shared resources and the pending demands. Finally, the third approach consists of performing a "good" program design so that no deadlock states are reached. This last approach, since it is done prior to the execution of the program, does not require the evolution of its execution to be monitored [2,11,5,14,8].

* This work is partially supported by the CICYT under grant TEL99-0582.

Our Work

In this paper, we focus on a potential approach for ensuring deadlock–freedom (due to critical section access) in a transparent manner from the programmer's point of view (i.e., it does not force programmers to pay attention to deadlocks when designing the program). That technique consists of using a number of operations to access some (artificially introduced) critical sections so as to guarantee that deadlock states will not be reached. Furthermore, that technique does not require monitoring program's executions.

In particular, we use a program transformation which introduces some new operations in the original program. As a result, it generates a deadlock–free "version" of the original program which maintains its behavior. In turn, it may result in a decrease of the program's concurrency. However, we think that there are many situations where programmers, in order to guarantee deadlock–freedom, may accept a concurrency reduction in their program (which, on the other hand, may be necessary if one wants to guarantee deadlock-freedom). We have used a well–known problem (that of the *dining philosophers*) to show how our approach behaves (see Section 5).

It must be noted that the only way to transform an existing program without additional information (neither from programmers nor from runtime values) consists of making use of *static analysis*. However, static analysis has several limitations which come from its lack of runtime data and from its exponential complexity. Consequently, the theoretical cost of our transformation are, in the worst case, non-tractable. We have tested the programs in the SPLASH II benchmark suite [13] using a "brute force" static analysis algorithm (see Section 2) and two of the applications were not analyzable due to lack of memory (on a personal computer Intel Pentium III with 1 Gigabyte of main memory).

Related Work

One of the first works dealing with static analysis of deadlocks is that of Taylor [15] where the author focus on blocking communications (*rendezvous*) in Ada programs.

Following [15], Masticola [11] presents some algorithms to detect deadlock–freedom in polynomial time. The work is also applied to Ada programs with blocking communications, but it is shown how these techniques can be applied to programs using binary semaphores to access shared resources. However, polynomial cost is attained by relaxing the knowledge of state reachability, and consequently it provides pessimistic solutions. Furthermore, neither [15] nor [11] provide any hint about how to ensure deadlock–freedom in the case where programs may become deadlocked. Static analysis of Java programs, following previous cited works, can be found in [6,12] (our work could benefit of the analysis presented in these references).

Maybe the only work (to the best of our knowledge) that focus on ensuring deadlock–freedom by using a prevention transformations is that of Taubin et al. [14]. By using their approach, the authors apply transformations to Petri net models to reduce the cost of deadlock states detection. Unfortunately, the overall cost remains exponential and it may occur that they are unable to provide any deadlock-free solution, even a pessimistic one.

In any case, we want to strength the goal of our work. The purpose is to point out the chance for automatic deadlock prevention techniques. The work is not specialized on static analysis of concurrent programs (it could complement those approaches) nor it takes into account all possible sources of deadlocks (i.e. accesses to replicated resources or cycles in data/control dependence as is the case of blocking communications).

The rest of the paper is organized as follows. In Section 2 we provide some definitions and we characterize when a program is deadlock–free. In Section 3, we introduce a transformation for safely removing one "source of deadlocks". Based on this transformation, in Section 4 we propose a program transformation algorithm which provides a deadlock–free version of a given program. An example of how our approach behaves is presented in Section 5. Finally, in Section 6 we present some concluding remarks.

2 Preliminaries

In this paper we use a simplified version of concurrent programs in which there are only operations to access critical sections: a $lock(cs)$ operation is used to acquire the critical section cs and an $unlock(cs)$ operation is used to release it. The use of these synchronization operations are common in the most popular languages used in concurrent programming, such as Java, Ada, C, etc. Operations are issued so that operations from the same process are executed sequentially. We use the relationship $\prec$ to characterize the order in which operations from the same process are intended to be executed. We also assume that a process can not request any critical section which it already locks, nor it can release any critical section which it does not lock.

Even though the model of programs seems quite restrictive, the conclusions drawn about deadlock prevention methods can also be used to reason about *structured programs* (i.e., programs that, besides sequential constructs, allow loops and conditionals). A program with conditional constructs is equivalent to a number of sequential programs covering all the possible paths that the different execution flows may follow. Such a number of sequential programs, even though it may be high, will be finite. Thus, it is enough to verify that every one of those programs is deadlock–free to ensure that the program (with conditional constructs) is deadlock–free. With regard to looping constructs, the situation is simpler if each acquired critical section within a given iteration is released in the same iteration[1]. That is because, for each one of the computations, the states corresponding to the process that performs the loop will be repeated in each one of the iterations, which allows us to treat loops as sequential constructs. A pre–processor (using a "brute force" algorithm) to obtain, given a real structured program, a set of programs such as those we consider, can be found in [1].

[1] A very reasonable assumption since, otherwise, it may be difficult to ensure that no critical section will be acquired/released more than once without it having been released/acquired meanwhile.

For our work, we will denote as op_{match} the unlock operation intended to release the critical section acquired by op (obviously op denotes a lock operation from the same process that op_{match}). Furthermore, we say an operation $op = lock(cs)$ is a *wrapper* of another operation op', denoted $op \in wrappers(op')$, if $op \prec op' \prec op_{match}$.

$$
\begin{array}{lllll}
P_1\colon & op_1 = lock(cs_1) & op_2 = lock(cs_2) & op_3 = unlock(cs_1) & op_4 = unlock(cs_2) \\
P_2\colon & op_5 = lock(cs_2) & op_6 = lock(cs_1) & op_7 = unlock(cs_2) & op_8 = unlock(cs_1)
\end{array}
$$

Fig. 1. A two–process program with 8 operations and 2 critical sections.

In Fig. 1, operations op_3 and op_4 are the respective matching operations to op_1 and op_2. Also, op_1 is a wrapper of op_2, which is in turn a wrapper of op_3.

Now, we introduce the "source of deadlock" (SoD) concept which is the key definition for characterizing deadlock-free programs. However, first we will introduce the "contemporariness" concept (on which the SoD definition is based). Roughly speaking, a set of operations is contemporary if it is possible an execution with a state in which all operations in the set are "active" at the same time.

Definition 1. *A set of operations OP of a program P is* contemporary *if there is at least one possible execution of P in which each operation in OP is the next operation to be executed in its respective process.*

It can be readily seen that operations op_2 and op_6 in Fig. 1 are contemporary. It can also be seen that operations op_3 and op_6 are not contemporary since both are in different accesses to critical section cs_2.

Definition 2. *Two disjoint sets of operations $OP = \{op_i\}_{i=1,\ldots,n,\ n>1}$ and $OP' = \{op'_i\}_{i=1,\ldots,n,\ n>1}$ form a* source of deadlocks (SoD) *if OP' is contemporary and $\forall op'_i = lock(cs)$ $(\exists op_i \in wrappers(op'_i)\ :\ op_{(i \bmod n)+1} = lock(cs))$.*

Note that this definition involves the existence of a set of critical sections $\{cs_i\}_{i=1,\ldots,n,\ n>1}$ in which, for each section, there is a lock in both sets, OP and OP'. Otherwise, the set OP' would not be contemporary. The sets $OP = \{op_1,\ op_5\}$ and $OP' = \{op_2,\ op_6\}$ in Fig. 1 form a SoD.

In [4], it is shown that programs are deadlock-free if and only if they do not have any SoD[2].

Theorem 1. *A program is deadlock–free iff it does not contain any SoD.*

2 There, the concept of *stopper* is used. This is equivalent to the definition of SoD given here.

3 A Transformation for Safely Removing One SoD

In this section it is proposed a program transformation which eliminates a SoD. Our approach consists of breaking, for a given SoD $\{OP, OP'\}$ in the original program, the contemporariness of operations in OP'. For such a task, we only have to ensure that at least a pair of operations in this set become non–contemporary. We achieve that by adding two pairs of operations that access a new (artificially introduced) critical section and that are wrappers of two different operations in OP'.

Transformation 1 *Given a program P with a SoD $d \equiv (OP,\ OP')$, we define the* non–contemporary transformation $NonContemporary(d,\ P)$ *as one which adds two accesses to a new defined critical section where each access contains just one (different) operation in OP'.*

We will call op_{prev} and op'_{prev} the lock operations introduced in the transformation $NonContemporary()$.

However, this transformation does not guarantee that new SoD's will not be induced. An example can be seen in Fig. 2 (b).

$$\begin{array}{l} P_1\!: lock(cs')\ \ \underline{lock(cs)}\ \ unlock(cs')\ \ \underline{lock(cs')}\ \ \ldots \\ P_2\!: \underline{lock(cs')}\ \ \underline{lock(cs)}\ \ \ldots \end{array}$$

(a)

$$\begin{array}{l} P_1\!: lock(cs')\ \ lock(cs)\ \ unlock(cs')\ \ \underline{lock(cs_{prev})}\ \ \underline{lock(cs')}\ \ unlock(cs_{prev})\ \ \ldots \\ P_2\!: \underline{lock(cs')}\ \ \underline{lock(cs_{prev})}\ \ lock(cs)\ \ unlock(cs_{prev})\ \ \ldots \end{array}$$

(b)

$$\begin{array}{l} P_1\!: lock(cs_{prev})\ \ lock(cs')\ \ lock(cs)\ \ unlock(cs')\ \ lock(cs')\ \ unlock(cs_{prev})\ \ \ldots \\ P_2\!: lock(cs_{prev})\ \ lock(cs')\ \ lock(cs)\ \ unlock(cs_{prev})\ \ \ldots \end{array}$$

(c)

Fig. 2. In Fig. (a), the underlined operations form a SoD d. After applying transformation $NonContemporary(d,\ P)$ (Fig. (b)), the former SoD no longer exists. However, the underlined operations form a new one. By also applying Lemma 1 (Fig. (c)), we ensure that new Sod's are not induced.

As the following lemma shows, if operations op_{prev} and op'_{prev} have not any wrapper, then no SoD is generated in the transformed program.

Lemma 1. *Let P be a program with a SoD $d \equiv (OP,\ OP')$ and let us apply $NonContemporary(d,\ P)$. If op_{prev} and op'_{prev} do not have any wrappers then no SoD is induced in the transformed program.*

Proof: By contradiction. Assume that program P' is the result of applying $NonContemporary(d,\ P)$. Assume also that op_{prev} and op'_{prev} do not have wrappers and that a new SoD $(OP'',\ OP''')$ has been generated.

Step 1: op_{prev} or op'_{prev} are included in $OP'' \cup OP'''$.
Proof: Let us assume that neither op_{prev} nor op'_{prev} are included in $OP'' \cup OP'''$. Thus:

1. All operations in $OP'' \cup OP'''$ are in P.
   Proof: Trivial since neither op_{prev} nor op'_{prev} are included in $OP'' \cup OP'''$.
2. The operations in OP''' are contemporary in P.
   Proof: As $(OP'',\ OP''')$ is a SoD in P', then OP''' is contemporary in P'. This set would also be contemporary in P since the $NonContemporary()$ only introduces new restrictions on the possibility of being contemporary.

Therefore, $(OP'',\ OP''')$ is a SoD in P, contradicting the assumption that it is a new SoD generated in the transformation.

Step 2: Either op_{prev} or op'_{prev} is included in OP'' and the other is included in OP'''.
Proof: Immediate given the definition of SoD and the fact that they are the only locks on the critical section defined in the transformation.

By Steps 1 and 2 we have that either op_{prev} or op'_{prev} is in OP'''. Furthermore, from the SoD's definition, all operations in OP''' have a wrapper. This contradicts the assumption that neither op_{prev} nor op'_{prev} have wrappers. Therefore, (OP'', OP''') is not a new SoD.

□

Fig. 2 (c) shows an example of how this result is applied. It must be noted that this is a simple example; in a general case only a part of a program is executed sequentially.

Remark. Whereas there can be several approaches for providing a program transformation capable to eliminate a given SoD, it has to be taken into account that such a transformation must ensure that new behaviors will not be introduced into the resulting program. Therefore, we can not use any technique based on making internal changes (such as moving an operation from one location to another) since they may not guarantee that the behavior of the program will be affected. Note that our approach only reduces the set of potential executions.

4 A Transformation for Making Programs Deadlock–Free

In this section we introduce a transformation which provides deadlock-free versions of concurrent programs in a transparent manner. Such a transformation, denoted $DirectTransform(P)$, is obtained by using $NonContemporary(d,\ P)$ and applying Lemma 1 directly. It uses the following functions:

– $GetSetofSoD(P)$: returns the SoDs of program P.
– $MoveBack(op, P)$: swaps the operation op with its immediately preceding one in the same process.

```
$DirectTransform(P)$::
  $D \leftarrow GetSetofSoD(P)$
  for each $d \in D$ do
    $NonContemporary(d, P)$
    while $wrappers(op_{prev}) \neq \emptyset$ do $MoveBack(op_{prev}, P)$
    while $wrappers(op'_{prev}) \neq \emptyset$ do $MoveBack(op'_{prev}, P)$
  return $P$
```

Fig. 3. Algorithm of $DirectTransform(P)$.

Fig. 3 shows the algorithm of this transformation. The next theorem proves that the program resulting from applying this algorithm is deadlock-free.

Theorem 2. *Given a program P, the program resulting from $DirectTransform(P)$ is deadlock-free.*

Proof:

Step 1: In each iteration of the **for** loop, a SoD is removed without generating a new one.
Proof: Immediate by Lemma 1. (Note that the number of iterations in the **while** loops is finite since, in the worst case, op_{prev} and op'_{prev} operations would be placed in the first position of the respective process).
Step 2: The algorithm ends.
Proof: Immediate, since the number of SoDs of a program with a finite number of operations is also finite.

By Steps 1 and 2, the transformed program is free of SoDs and by Theorem 1, it is also deadlock-free.

$\square$

Regarding the complexity of $DirectTransform()$, it must be taken into account that function $GetSetofSoD()$ has non-polynomial cost. That is because it is hindered by the well–known state explosion problem: the number of states in a concurrent system tends to increase exponentially with the number of processes [15]. However, this problem can be solved by not checking the contemporariness of OP' set (obviously, obtaining an approximate result[3]). In that case, the cost of function $GetSetofSoD()$ is to $O(n + p^2c^4 + p^3)$, being n the number

[3] The function, in this case, returns all SoDs but can also return some false SoDs.

of operations of the program, p the number of processes and c the number of critical sections used.

Whereas here we have introduced a transformation that directly applies Lemma 1, other transformation can also be introduced. For instance, it can be defined a new transformation that, whenever a SoD is eliminated, the operations op_{prev} and op'_{prev} are selectively moved back, applying Lemma 1 only in the worst case. This transformation would be more costly to apply (to eliminate a SoD it must be checked that new SoD's are not induced) but the resulting program may have a lower reduction of its concurrency.

5 An Example

As it has been said previously, a drawback of applying the above mentioned transformations is that they restrict the way in which processes can execute their operations. Therefore, programs may suffer a loss on the number of states the program can reach (from now called *potential concurrency*), which may become in a loss on the program's performance. In this section, we use the widely know *dining philosophers* problem to show how our proposed transformations affect the concurrency.

For our comparison we will take, first, the "classical" solution. By using this solution, each philosopher, say philosopher i, locks sections cs_i and $cs_{(i \bmod n)+1})$ in order, except for one of the philosophers that operates in the opposite way. It is known that this solution guarantees that philosophers will not get deadlocked.

On the other hand, we also use $DirectTransform()$ to obtain a deadlock–free version for the philosophers problem. As Fig. 4 shows, the version obtained by using $DirectTransform()$ has not a significant concurrency loss with respect to the classical algorithm. However, whereas the classical solution burdens programmers with a new task (i.e., finding that solution), by using one of our transformation the algorithm is obtained in a transparent manner from the programmer's point of view.

6 Conclusion and Future Work

In this paper we have introduced a method that provide deadlock-free versions of concurrent programs in a transparent manner from the programmer's point of view. Even though the cost of the transformation is exponential, it can be drastically reduced using a pessimistic approach.

There are some other issues, which we are currently working in, that need further research. The decrease in concurrency that $DirectTrasform()$ provides depends on which operations are chosen to be synchronized in $NonContemporary()$. So, it will be worthwhile to try different combinations. Also, there are tools as [5,6] and [9] that can be used to check different program properties as deadlock–freedom. So, it will be worth to study how integrate our prevention techniques in such programming environments.

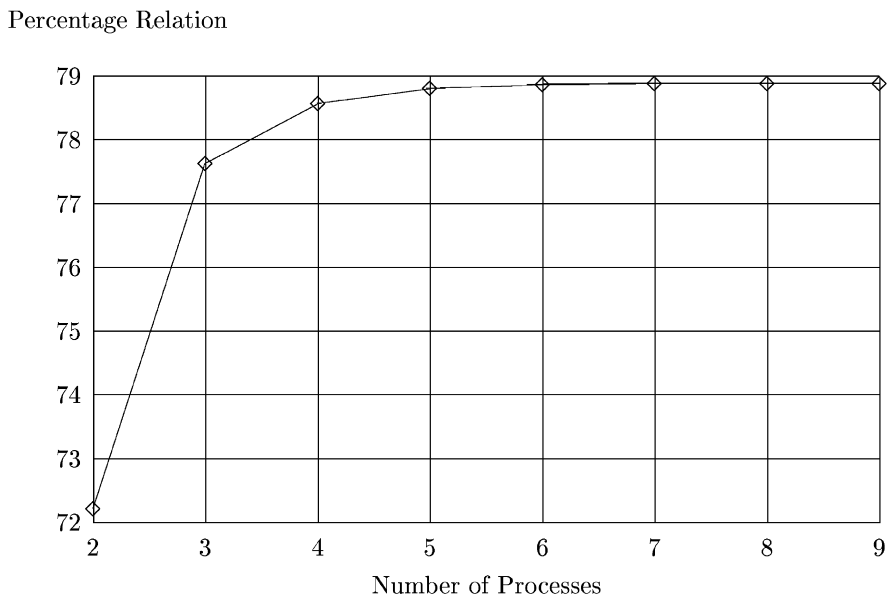

Fig. 4. Relationship between the potential concurrency provided by using *DirectTransform*() and the "classical" solution in the *dining philosophers* problem. Only one iteration for each philosopher has been considered.

References

1. L. Almajano. Desarrollo de un preprocesador para el an lisis est tico de programas concurrentes, 2001. Proyecto final de carrera para la obtenci n del t tulo de Ingeniero en Inform tica (Universidad Jaume I).
2. Ö. Babaoglu, E. Fromentin, and M. Raynal. A unified framework for the specification and run-time detection of dynamic properties in distributed computations. Technical Report UBLCS-95-3, Department of Computer Science, University of Bologna, February 1995.
3. F. Belik. An efficient deadlock avoidance technique. *IEEE Transactions on Computers*, 39(7):882–888, July 1990.
4. V. Cholvi and P. Boronat. A minimal property for characterizing deadlock-free programs. *Information Processing Letters*, (77):283–290, 2001.
5. J.C. Corbett. Constructing compact models of concurrent java programs. In Michal Young, editor, *ISSTA 98: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–10, 1998.
6. J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
7. J.R. Gonz lez de Mend vil, J.M. Bernab u, A. Demaille, and J.R. Garitagoitia. A distributed deadlock resolution algorithm for the and request model. *IEEE Transactions on Parallel and Distributed Systems*, 1999.

8. C. Flanagan and M. Abadi. Types for safe locking. In *ESOP 99: European Symposium on Programming, LNCS 1576*, pages 91–108, 1999.
9. Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
10. T.F. Leibfried. A deadlock detection and recovery algorithm using the formalism of a directed graph matrix. *Communications of the ACM*, 23(2):45–55, April 1989.
11. S.P. Masticola. *Static detection of deadlocks in polynomial time.* PhD thesis, The State University of New Jersey, May 1993.
12. G. Naumovich, G.S. Avrunin, and L.A. Clarke. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *ACM SIGSOFT Software Engineering Notes, Proceedings of the 7th European Engineering*, volume 24, 1999.
13. J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.
14. A. Taubin, A. Kondratyev, and M. Kishinevsky. Deadlock Prevention Using Petri Nets and their Unfoldings. Technical Report 97-2-004, Department of Computer Hardware, The University of Aizu, March 1997.
15. R.N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26:362–376, 1983.