Information Processing Letters 77 (2001) 283–290

Information Processing Letters

www.elsevier.com/locate/ipl

A minimal property for characterizing deadlock-free programs [☆]

Vicent Cholvi [*], Pablo Boronat

Departament d'Informàtica, Universitat Jaume I, 12071, Castelló, Spain

Received 1 August 1999
Communicated by J.L. Fiadeiro

Abstract

A fundamental issue in the development of concurrent programs is the resource allocation problem. Roughly speaking, it consists of providing some mechanism to avoid race conditions in the access of shared resources by two or more concurrent processes. For such a task, maybe the most widely mechanism consists of using critical sections.

Unfortunately, it is also widely-known that programs which use several critical sections may suffer from deadlocks. In this paper, we identify a program property, namely, being *stopper-free*, which can be used to know if programs are deadlock-free. Indeed, since we have proved that programs are deadlock-free if and only if they do not have any stopper, thus looking for a stopper is equivalent to identifying a situation where a program may suffer a deadlock. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Program properties; Concurrency; Distributed computing; Static analysis; Deadlocks; Theory of computation

1. Introduction

In the development of concurrent programs, considerable effort has been devoted to study the resource allocation problem, where by resource we mean a physical device as well as a section of code. While local resources are accessed only by one process, shared resources can be accessed by many processes. If two or more processes simultaneously contend for the same resource, they can leave the resource in an undefined state (Fig. 1(a)). Therefore, it is necessary some mechanism to avoid "race conditions".

Maybe the most widely used mechanism for such a task consists of using *critical sections* [1]. Roughly speaking, a critical section can be seen as a "section of code" which guarantees processes exclusive access to the resources allocated within. In order to access a critical section a process must firstly acquire it (by executing a separate section of code), releasing it (by executing another section of code) after leaving that critical section. Obviously and since that forces shared resources to be "sequentially" accessed, critical sections are an effective way of avoiding race conditions (Fig. 1(b)).

Unfortunately, it is also widely-known that programs which use several critical sections may suffer from deadlocks (i.e., the program being in an infinite wait) [7]. Fundamental to this problem is knowledge of situations where deadlocks may occur.

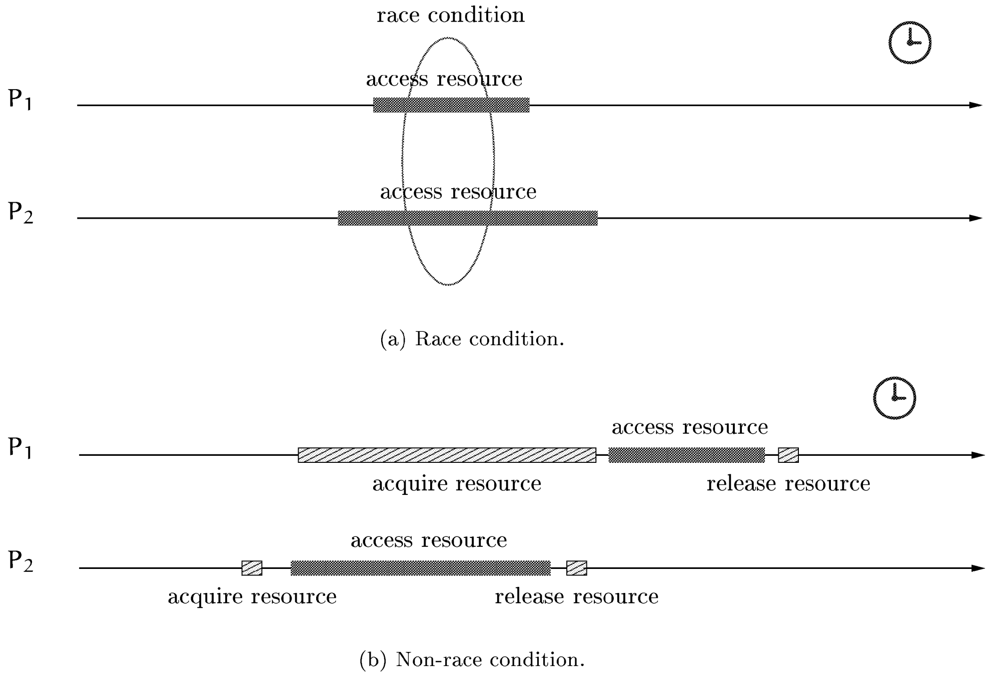
Detecting when the state of a distributed computation satisfies a certain property, namely, if deadlocks may occur, constitutes a fundamental issue in the design of concurrent programs. Traditionally and in or-

[☆] This work is partially supported by the CICYT under grant TEL99-0582.

[*] Corresponding author.

E-mail addresses: vcholvi@inf.uji.es (V. Cholvi), boronat@inf.uji.es (P. Boronat).

0020-0190/01/$ – see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S0020-0190(00)00158-7



(a) Race condition.

(b) Non-race condition.

Fig. 1. Accessing a shared resource.

der to detect if a concurrent program holds a certain property, three possible approaches have been used: during the program's execution, after the program's execution or prior to any program's execution [10].

The first approach (also called *dynamic analysis*) must be verified by monitoring the execution as it evolves [2]. However and even though several efficient algorithms have been developed to detect deadlocks dynamically [4,5], conclusions that they may draw are not about all possible executions of the program but about an actual execution. Similarly, the second approach (also called *post-mortem* analysis) is unsuitable for our purposes, since it requires the program to end, which can only happen if the program's execution has not been deadlocked (which is precisely the studied property). Finally, the third approach (also called *static analysis*) is performed on a model of the program without requiring test executions. In this paper, we identify a program property which can be used to known if programs are deadlock-free by using static analysis. Such a property is minimal in the sense that programs are deadlock-free if and only if they do not satisfy it.

The rest of the paper is organized as follows. In Section 2 we define the model used throughout the paper. In Section 3 we identify a program property which meets the above mentioned criteria. Finally, in Section 4, we talk about the applicability of our result and discuss some current work.

2. Formal framework

2.1. Program specification

The first thing we do in this section is to provide a definition of program. A *program* consists of a set of operations from a set of processes such that operations from the same process are intended to be issued in a fixed way. In order to characterize that behavior, we will use an (irreflexive) order $\prec$ which totally orders operations from the same process.

An operation may be either internal to a process and cause only a local state change, or it may involve interaction with other processes by mean of using

State:

for each process $p \in \mathit{Set\_of\_Processes}$:

$\mathit{status}(p)$: status of process p, consisting of a pair whose first component is either *lock* or *unlock* and the second component a critical section; initially the special value *idle*.

for each critical section $cs \in \mathit{Set\_of\_Critical\_Sections}$:

$\mathit{block}(cs)$: process blocking cs; initially the special value *none*.

Actions:

Input $\mathit{lock}_p^{\mathrm{start}}(cs)$
EFFECT:
$\mathit{status}(p) \leftarrow \langle \mathit{lock}, cs\rangle$

Output $\mathit{lock}_p^{\mathrm{end}}(cs)$
PRECONDITION:
$\mathit{status}(p) = \langle \mathit{lock}, cs\rangle$
$\mathit{block}(cs) = \mathit{none}$
EFFECT:
$\mathit{block}(cs) \leftarrow p$
$\mathit{status}(p) \leftarrow \mathit{idle}$

Input $\mathit{unlock}_p^{\mathrm{start}}(cs)$
EFFECT:
$\mathit{status}(p) \leftarrow \langle \mathit{unlock}, cs\rangle$

Output $\mathit{unlock}_p^{\mathrm{end}}(cs)$
PRECONDITION:
$\mathit{status}(p) = \langle \mathit{unlock}, cs\rangle$
EFFECT:
$\mathit{block}(cs) \leftarrow \mathit{none}$
$\mathit{status}(p) \leftarrow \mathit{idle}$

Scheme 1.

shared variables. However, here we consider only operations for accessing shared resources.

A lock operation issued by process p, denoted $\mathit{lock}_p(cs)$, is used to acquire the critical section cs. Similarly, an unlock operation, denoted $\mathit{unlock}_p(cs)$, is used by process p to release such a critical section. In this paper, we do not allow any process to request for any critical section which it already locks, nor release any critical section which it does not lock.

2.2. *System_Machine specification*

Now, we provide a formal description of the protocol for accessing critical sections (i.e., the protocol that implements lock and unlock operations). For such a task, we use a specification based on a state machine, *System_Machine*, for which it has been used a model based on the I/O automata model of Lynch and Tuttle (see the Appendix A).

In *System_Machine*, each operation *op* (either a lock or an unlock) is modeled by using two actions, which represent, respectively, its start ($\mathit{start}(op)$) and end ($\mathit{end}(op)$). *System_Machine* is parameterized by a given set of processes *Set_of_Processes* and a given set of critical sections *Set_of_Critical_Sections*. The formal definition of *System_Machine* is given in Scheme 1.

2.3. *Interaction between a program and System_Machine*

In our formalism, the result of computing a program P on a system specified by *System_Machine* consists on an execution of *System_Machine* where, for each operation *op* in P, there is an input action $\mathit{start}(op)$ which is immediately followed by its corresponding output action $\mathit{end}(op)$. That restricts the set of executions to those that follow the model of sequential processes executing blocking operations: the process executing an operation is forced to block until the operation completes.

Definition 1. A *computation* of a program P under a system modeled by *System_Machine* consists of an execution of the automata *System_Machine* such that its input actions correspond with operations of P, they preserve the order $\prec$ and none of them is enabled at state S by process p if $S.\mathit{status}(p) \neq \mathit{idle}$.

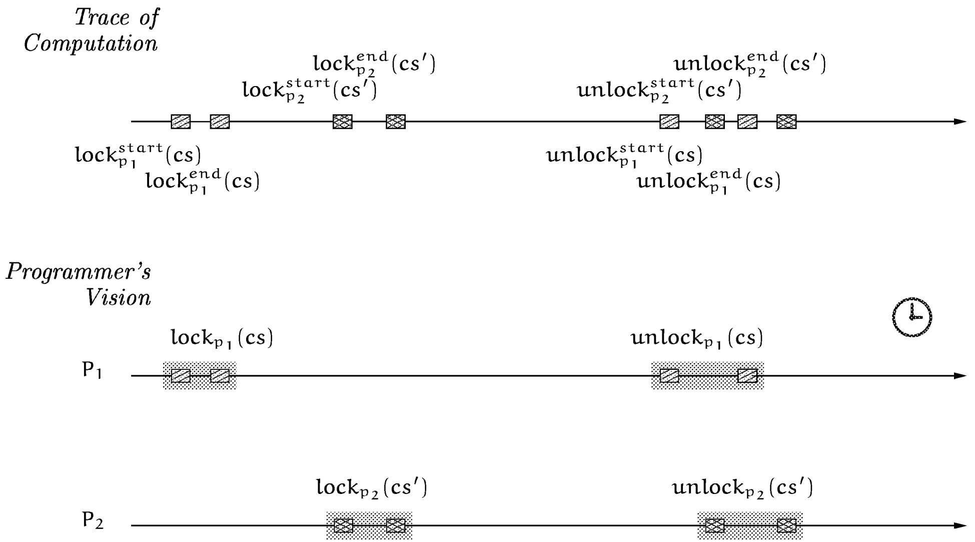


Fig. 2. Relationship between the trace of a computation and a programmer's vision of such a trace.

Note that, for this definition to make sense, the set of processes and critical sections of *System_Machine* are assumed that respectively include the set of processes and critical sections of program P.

Fig. 2 shows a graphical representation of the trace of a given computation (i.e., the subsequence of the computation consisting of external actions only), where the horizontal line represents the execution order of those actions progressing from left to right. We also provide a "programmer's vision" of such a trace with actions from the same process explicitly laid out and transformed into operations. Notice that our formalism does not make use of time. However and as it can be seen, using a "space-time" diagram is a convenient tool to visualize distributed computation. In what follows and for the sake of simplicity we will talk simply of programs, omitting that such programs are computed on a system modeled by *System_Machine*.

2.4. Deadlock specification

As it has been said, a major contribution of this paper consists of identifying a program property for characterizing deadlock-free programs. Thus, we have to define what is a deadlocked computation.

Definition 2. A computation α of a program P has a *deadlock* at state S if

$$\exists\{cs_i\}_{\substack{i=1,\dots,n\\ n>1}} \quad \text{and} \quad \exists\{p_i\}_{\substack{i=1,\dots,n\\ n>1}} \text{ such that}$$
$$\forall i\ \big(S.block(cs_i) = p_i \quad \text{and}$$
$$S.status(p_i) = \langle lock, cs_{(i \bmod n)+1}\rangle\big).$$

Fig. 3 shows a programmer's vision of a given computation which at state S has a deadlock (i.e., it is deadlocked).

At this point, it is worthwhile to note that, even though we use a notion of program where operations from the same process are intended to be issued in a fixed way, for the purpose of detecting whether a program is deadlock-free, conclusions drawn about them can be used to reason about *structured programs* (i.e., programs that, besides sequential constructs, allow loopings and conditionals). Indeed, a program with conditional constructs is equivalent (for our purpose) 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 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

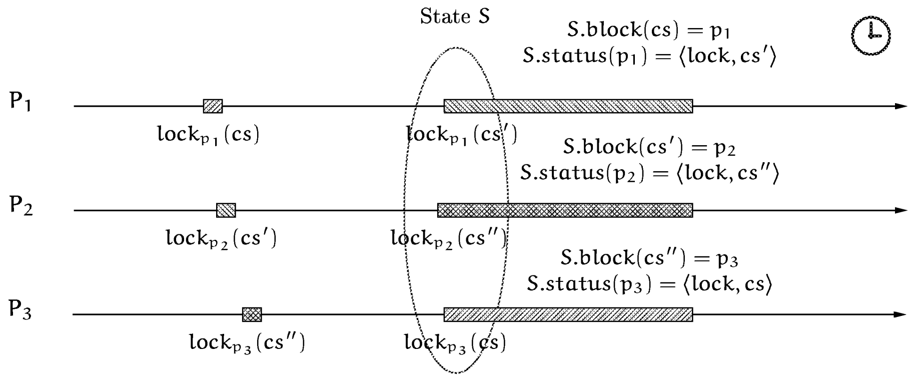


Fig. 3. Deadlocked computation.

to looping constructs, the situation is simpler if each acquired/released critical section within a given loop is released/acquired in the same loop[1]. That is because, for each one computation, the states corresponding to the process that performs the loop will be repeated in each one iteration. Therefore, for the purpose of detecting whether there is a deadlock or not, it will be equivalent to a sequential construct.

3. Stoppers: A property for characterizing deadlocks

In this section, we introduce the "stopper" concept which is the key definition for characterizing deadlock-free programs. However, first we will introduce the "contemporariness" concept (on which the stopper concept is based on). For such a task, we use the next notation:

Notation 1. Let *op* be a program operation used to acquire a given critical section. We denote op_{match} the unlock operation (from the same process) intended to release the critical section acquired by *op*.

Notation 2. We say an operation $op = lock_p(cs)$ is *wrapper* of another operation op', denoted $op = wrapper^{cs}(op')$, if $op \prec op' \prec op_{\text{match}}$. We also denote $op \in wrappers(op')$ if, for some critical section, *op* is wrapper of op'.

Definition 3. Let OP be a set of operations, each one from a different process, of a program P and let P_{OP} denote a subprogram of P such that each one operation is previous (with respect to $\prec$) to some operation in OP. We say OP is *contemporary* if there is a total ordering $\prec^{\mathrm{T}}$ on the operations in P_{OP} that preserves $\prec$ such that if $op' \prec^{\mathrm{T}} op$ and $op = lock_p(cs)$ and $op' = lock_{p'}(cs)$ (where $p \neq p'$) then $op'_{\text{match}} \prec^{\mathrm{T}} op$.

Fig. 4 shows the operations that form a program (explicitly laid out for each process). As it can be readily seen, boxed operations are contemporary since there is a total ordering for their preceding operations which preserves the conditions stated at Definition 3 (we provide a possible ordering).

As the next lemma shows, contemporary operations are characterized by the fact that, in some computation, they may be *outstanding* (i.e., at some state of the computation, the start actions have been executed and the end actions have not been executed).

Lemma 1. *Let* OP *be a set of operations, each one from a different process, of a program* P. OP *is contemporary iff there is a computation of such a program where at a given state, the whole set of operations in* OP *are outstanding.*

Proof. ($\Rightarrow$) As OP is contemporary, by Definition 3, there is a total ordering $\prec^{\mathrm{T}}$ on the operations in P_{OP}

[1] A very reasonable assumption since, otherwise, it may be difficult to ensure that none critical section will be acquired more than once and vice versa.

$$\mathsf{P}_1 : \mathsf{lock}_{p_1}(cs'')\ \mathsf{lock}_{p_1}(cs')\ \mathsf{unlock}_{p_1}(cs'')\ \mathsf{unlock}_{p_1}(cs')\ \boxed{\mathsf{lock}_{p_1}(cs'')}\ \ldots$$

$$\mathsf{P}_2 : \mathsf{lock}_{p_2}(cs)\ \mathsf{unlock}_{p_2}(cs)\ \boxed{\mathsf{lock}_{p_2}(cs'')}\ \ldots$$

$$\mathsf{P}_3 : \mathsf{lock}_{p_3}(cs)\ \boxed{\mathsf{lock}_{p_3}(cs')}\ \ldots$$

$$\text{Order: } \mathsf{lock}_{p_1}(cs'')\ \mathsf{lock}_{p_1}(cs')\ \mathsf{unlock}_{p_1}(cs'')\ \mathsf{unlock}_{p_1}(cs')\ \mathsf{lock}_{p_2}(cs)\ \mathsf{unlock}_{p_2}(cs)\ \mathsf{lock}_{p_3}(cs)\ \ldots$$

Fig. 4. Contemporary operations.

that gives a way for executing those operations so that all of them end.

Let us now compute the whole program P by executing first the operations in P_{OP} in the ordering given by $\prec^{\mathrm{T}}$. At such a state, as operations in P_{OP} will end, nothing prevents operations in OP from starting immediately one after another. That makes operations in OP being outstanding.

($\Leftarrow$) Assume a given computation where, at a given state, operations in OP are outstanding. Therefore, at that state operations in P_{OP} have ended.

It is immediate to verify that the execution order of such operations fulfills the specification of $\prec^{\mathrm{T}}$ in Definition 3. □

Now, we can proceed with the definition of stopper.

Definition 4. Given a program P, we say that a set of critical sections $\{cs_i\}_{i=1,\ldots,n,\ n>1}$ forms a *stopper* if $\exists\{p_i\}_{i=1,\ldots,n,\ n>1}$ such that

$$\forall i\ \big(\exists op, op' \colon op = wrapper^{cs_i}(op') \quad \text{and} \quad op' = lock_{p_i}(cs_{(i \bmod n)+1})\big)$$

and where the set formed by those op' operations is contemporary.

If we remove the contemporariness condition for the op' operations in the last definition, we say that those critical sections form a *potential-stopper*.

The next theorem states our main result, namely, that programs are deadlock-free if and only if they do not have any stopper.

Theorem 1. *Let stoppers*(P) *denote the whole set of stoppers for a program* P*. Then,* P *is deadlock-free iff stoppers*(P) $= \emptyset$.

Proof. ($\Rightarrow$) By contradiction.

Let P be a program with a stopper. We will prove that there is at least a computation of P with a deadlock.

From the stopper's definition, we know that there is a set of critical sections $\{cs_i\}_{i=1,\ldots,n,\ n>1}$ and a set of processes $\{p_i\}_{i=1,\ldots,n,\ n>1}$ such that

$$\forall i\ \big(\exists op, op' \colon op = wrapper^{cs_i}(op') \quad \text{and} \quad op' = lock_{p_i}(cs_{(i \bmod n)+1})\big)$$

and where the set formed by those op' operations is contemporary.

By Lemma 1, there is a computation α of P such that at a given state S, the whole set of op' operations are outstanding.

We will prove that at this state, α has a deadlock; that is, for all i the following holds:

(1) $S.block(cs_i) = p_i$.

Proof. We know that at state S all the op' operations are outstanding. Take

$$op' = lock_{p_i}\big(cs_{(i \bmod n)+1}\big).$$

We also know that there is an op operation such that $op = wrapper^{cs_i}(op')$.

Therefore, from the specification of *System_ Machine* (and taking into account the wrapper's definition), we have that $S.block(cs_i) = p_i$. □

(2) $S.status(p_i) = \langle lock, cs_{(i \bmod n)+1}\rangle$.

Proof. We know that at state S all the op' operations are outstanding. Take

$$op' = lock_{p_i}\big(cs_{(i \bmod n)+1}\big).$$

Therefore, from the specification of *System_Machine*, we have that

$$S.status(p_i) = \big\langle lock, cs_{(i \bmod n)+1}\big\rangle. \qquad \square$$

($\Leftarrow$) By contradiction.

Let α be a computation of P which has a deadlock at state S. We will prove that $stoppers(\mathsf{P}) \neq \emptyset$.

By definition of deadlock, we know that at some state S there is a set of critical sections $\{cs_i\}_{\substack{i=1,\dots,n\\ n>1}}$ and a set of processes $\{p_i\}_{i=1,\dots,n,\ n>1}$ such that

$$\forall i\ \big(S.block(cs_i) = p_i \quad \text{and} \quad S.status(p_i) = \langle lock, cs_{(i \bmod n)+1}\rangle\big).$$

We will prove that such critical sections form a stopper; that is, we will prove the following two things:

(1) For all i, there is a pair of operations op and op' such that the following holds:

(a) $op = wrapper^{cs_i}(op')$.

Proof. By the conditions of deadlock, we have that $S.status(p_i) = \langle lock, cs_{(i \bmod n)+1}\rangle$ and $S.block(cs_i) = p_i$.
Therefore, from the specification of *System_Machine* and taking into account the definition of computation, we have that

$$\exists op = lock_{p_i}(cs_i)\colon\ op \prec op' \prec op_{\text{match}}. \qquad \square$$

(b) $op' = lock_{p_i}(cs_{(i \bmod n)+1})$.

Proof. By the conditions of deadlock, we have that $S.status(p_i) = \langle lock, cs_{(i \bmod n)+1}\rangle$. Therefore, from the specification of *System_Machine* and taking into account the definition of computation, we have that

$$op' = lock_{p_i}\big(cs_{(i \bmod n)+1}\big). \qquad \square$$

(2) The set formed by the previous op' operations is contemporary.

Proof. By contradiction. Assume it is not contemporary. By Lemma 1, such operations can not be outstanding at the same time. Therefore, for some i, we have that $S.status(p_i) \neq \langle lock, cs_{(i \bmod n)+1}\rangle$.

However, we have assumed that at state S there is a deadlock, which implies that $\forall i\ (S.status(p_i) = \langle lock, cs_{(i \bmod n)+1}\rangle)$, thus contradicting the hypothesis. $\square$

4. Discussion and current work

In order to check whether or not a program is deadlock-free by using static analysis, the simplest technique consists of "enumerating" the whole set of states where a given program is executed and search for deadlock states. Unfortunately, such a technique 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. In fact, because of the related complexity results, static analysis tools are necessarily exponential on the number of processes [6] and so does finding if a program is stopper-free.

Despite that result that shows the intractability (in general) of using a static analysis to check whether or not a program is deadlock-free, there are some reasons for supporting such an approach. Indeed, to check whether or not a program is stopper-free, first we look for any potential-stopper. Therefore, if a program has not any potential-stopper (which is a relatively simple task), we can say it is deadlock-free. Nevertheless, despite there are many potential-stopper-free programs for which this last result is relevant (see for instance the SPLASH testbed [9]), there are other programs for which this situation do not apply. In that case, it is necessary to check whether the potential-stoppers are in fact stoppers by identifying if the op' operations in Definition 4 are contemporary, and this is not a simple task.

At this point, we are currently focusing our efforts in guaranteeing deadlock-freedom. Our approach consists of making op' operations in Definition 4 being non-contemporary (note that, whereas this approach is not surprising from our results, here we clearly showed the rationale behind it). For such a task, we are following two different approaches: the first one consists of introducing new locks to ensure that their subsequent operations will not be executed at the same time; the second one consists of introducing some dependencies between existing locks so that they not only will check that no other process is accessing the requested critical section but that some other critical sections are not being accessed at this time [3].

Appendix A. The I/O automata formalism

In this Appendix we introduce the I/O automata formalism. We use a slight simplification of the I/O automaton of Lynch and Tuttle, ignoring the aspects related to liveness. We include only those parts we consider necessary to understand the paper. For a full discussion, the reader is referred to [8].

In the I/O automata formalism, all components in a system are modeled by using automata. An I/O *automaton* $\mathcal{A}$ is composed of:

(1) A set of *states*, some of which are designated as *initial states*.
(2) A *signature* of actions, $sig(\mathcal{A})$. Such a signature consists of three mutually disjoint sets of *actions*: *input*, $in(sig(\mathcal{A}))$; *output*, $out(sig(\mathcal{A}))$; and *internal*, $int(sig(\mathcal{A}))$. We denote the set of external actions of the signature as $ext(sig(\mathcal{A})) = in(sig(\mathcal{A})) \cup out(sig(\mathcal{A}))$.
(3) A *transition relation*, which is a set of triples of the form $(\mathsf{S}, \pi, \mathsf{S}')$, where S' and S are states, and π an action. This triple means that in state S, the automaton can atomically do action π and change to state S'.

An element of the transition relation is called a *step*. Output actions are intended to model the actions that are triggered by the automaton itself, while input actions model actions that are triggered by the environment of the automaton (an automaton must be prepared to receive any input action at any time). Internal actions are used to model communication between components within the automaton.

An *execution* α of $\mathcal{A}$ is a (finite or infinite) alternating sequence $\mathsf{S}_0\pi_1\mathsf{S}_1\pi_2\ldots\pi_n\mathsf{S}_n\ldots$ of states and actions of $\mathcal{A}$, beginning with an initial state, and (if finite) ending with a state. We denote the set of executions of $\mathcal{A}$ by $execs(\mathcal{A})$. From an execution α, we can extract the *trace*, which is the subsequence of the execution consisting of external actions only. Because transitions to different states may have the same action, different executions may have the same trace. We denote the set of traces of $\mathcal{A}$ by $traces(\mathcal{A})$.

References

[1] G.R. Andrews, Concurrent Programming. Principles and Practice, Benjamin/Cummings Publishing Company, Inc., New York, 1991.
[2] Ö. Babaoglu, E. Fromentin, 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] P. Boronat, V. Cholvi, Dependences between critical sections in synchronized memory models, in: Proc. International Conference on Computing and Information, June 1998.
[4] G. Bracha, S. Toueg, Distributed deadlock detection, Distributed Comput. 2 (3) (1987) 127–138.
[5] P.A. Buhr, M. Fortier, M.H. Coffin, Monitor classification, ACM Comput. Surveys 27 (1) (1995) 63–107.
[6] J.C. Corbett, Evaluating deadlock detection methods for concurrent software, IEEE Trans. Software Engrg. 22 (3) (1996).
[7] E.G. Coffman, M.J. Elphick, A. Shoshani, System deadlocks, ACM Comput. Surveys 3 (2) (1971) 67–78.
[8] N. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1996.
[9] J. Singh, W. Weber, A. Gupta, SPLASH: Stanford parallel applications for shared memory, Comput. Architecture News 20 (1) (1992) 5–44.
[10] R.N. Taylor, A general-purpose algorithm for analyzing concurrent programs, Comm. ACM 26 (1983) 362–376.