Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 113 (2005) 65–83

www.elsevier.com/locate/entcs

Enforcing Concurrent Temporal Behaviors

Doron Peled and Hongyang Qu

Department of Computer Science
The University of Warwick
Coventry, CV4 7AL, UK

Abstract

The outcome of verifying software is often a ‘counterexample’, i.e., a listing of the actions and states of a behavior not satisfying the specification. In order to understand the reason for the failure it is often required to test such an execution against the actual code. In this way we also find out whether we have a genuine error or a “false negative”. Due to nondeterminism in concurrent code, recovering an erroneous behavior on the actual program is not guaranteed even if no abstraction was made and we start the execution with the prescribed initial state. Testers are faced with a similar problem when they have to show that a suspicious scenario can actually be executed. Such a scenario may involve some intricate scheduling and thus be illusive to demonstrate. We describe here a program transformation that translates a program in such a way that it can be verified and then reverse transformed for testing a suspicious behavior. Since the transformation implies changes to the original code, we strive to minimize its effect on the original program.

Keywords: Behavior monitoring, Concurrency, Counterexample analysis, Model Checking, Nondeterminism, Temporal Logic, Testing.

1 Introduction

Verification is used to pinpoint the existence of errors in software. The outcome of the verification process is often given by listing the sequence of atomic actions and global states comprising a behavior not satisfying the specification, called a *counterexample*. Since it is often not the code itself that is being verified, but rather a model of it, there is a non negligible likelihood of encountering a ‘false negative’. That is, an execution that does not conform with a behavior of the actual program. Since false negatives occur frequently, an execution suspicious of being faulty needs to be carefully checked. Unfortunately, executions of concurrent programs may be quite long and complicated when manually analyzed. Testers face a similar problem when they are required to show that some suspicious behavior actually occurs during some run of the code. Given a behavior that appears under

doi:10.1016/j.entcs.2004.01.034

some uncommon scheduling, it may be very hard for a tester to enforce the tested program to execute it.

We are interested here in causing the checked code to behave according to a given suspicious execution, which may be the result of a verification or testing effort. We want to be able to reconstruct and inspect this behavior in the context of the tested or verified code. Due to nondeterminism associated with concurrently executing events, we are not guaranteed to recover the given execution without enforcing some modification to the checked software. We therefore concentrate on minimizing the effect of the changes to the original code. We suggest a simple and automatic transformation that can be applied to the code in order to recover the suspicious behavior. Our method gives the verification engineer or tester a tool for checking and demonstrating the existence of the error in the code. We impose the following constraints on the suggested software transformation:

- Minimize the changes to the software. We want to deviate as little as possible from the original program.
- Enforce the required execution exactly when choosing an appropriate initial state. Other executions are still available when this initial state is not selected.
- Any concurrency or independence between executed actions is preserved. We are reconstructing a concurrent execution, rather than a completely synchronized one in which one action is executed at a time. Hence our solution is *distributed* rather than centralized.
- Preserve the checked property. Not all the execution sequences with the same concurrent structure as the original counterexample necessarily satisfy the same temporal properties.
- Apply the construction to a finite representation of infinite execution, i.e., an ultimately periodic sequence.

Although our transformation will be demonstrated for a given (Pascal-like) syntax, it is language-independent. We discuss some language and implementation issues in Section 7.

Our architecture is shown in Figure 1. In order to perform the verification, the program is translated into a finite set of atomic actions. This translation may also include an abstraction that simplifies the verified model. Another component of the translation is a dependency relation, which includes pairs of actions that cannot be executed concurrently, e.g., due to the use of a shared variable or a message queue. Finally, a third component of the translation is an annotated code, which has pointers to various locations in the text, specifically the beginning and the end of the text of the code related to atomically executed actions. These pointers are useful in adding new instructions (or changing the existing ones) in a way that will enforce executing the suspicious behavior.

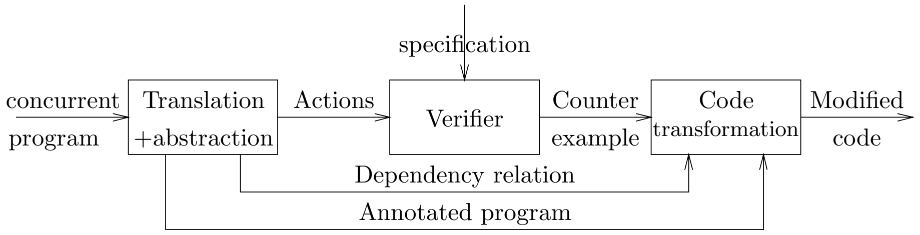

Fig. 1. System Architecture

Some related work focuses on creating test cases for message sequence charts. In [9], the difficulty of constructing such test cases is demonstrated. (But note that the model of message sequence charts is somewhat different, as it allows all the processes to *decide* globally to behave according to one scenario or another; this is a peculiarity of that model that itself requires analysis [2].) In [15], a solution that is based on adding coordinating messages is presented. Our solution differs as it does not impose changes that restrict the concurrency of the system. Transformations for monitoring the execution of concurrent programs are suggested in [13]. Our contribution is in analyzing such transformations in the framework of partial order semantics and equivalence between execution sequences. We also show how to handle ultimately periodic sequences (monitored for a finite, but not apriory bounded, length of time), and how to preserve the checked property.

2 Preliminaries

As a running example, consider Dekker's solution to the mutual exclusion algorithm in Figure 2. The flow graph appears in Figure 3. This graph was obtained automatically from the code using the PET testing system [7]. Suppose we start the execution with $turn = 1$. The following execution σ_1 can be obtained, where process $P1$ enters its critical section. Each line represents the occurrence of an action. It consists of a sequence number, the execution process, followed by the number of the flow chart node involved (in parentheses) according to Figure 3, and followed by the text corresponding to the action. An action corresponding to a condition is also followed by a 'yes' or a 'no', depending on whether the test succeeds or fails.

Here, both processes proceed to signal that they want to enter their critical sections, by setting $c1$ and $c2$ to 0 (lines 7 and 8), respectively. Because $turn = 1$ ($turn$ is checked in lines 11 and 12), process $P1$ has priority over process $P2$. This means that process $P2$ gives up its attempt, by setting $c2$ to 1 (line 13), while process $P1$ insists, waiting for $c2$ to become 1 (checked in line 14) and then enters its critical section (line 15).

```
1: (P1(0) : start)
2:   (P2(0) : start)
3: [P1(1) : c1:=1]
```

```
                    boolean c1, c2 ;
                    integer (1..2) turn;

P1::c1:=1;                            P2::c2:=1;
  while true do                         while true do
  begin                                 begin
    c1:=0;                                c2:=0;
    while c2=0 do                         while c1=0 do
      begin                                 begin
        if turn=2 then                        if turn=1 then
          begin                                 begin
            c1:=1;                                c2:=1;
            while turn=2 do                       while turn=1 do
             begin                                 begin
              /* no-op */                           /* no-op */
             end;                                  end;
            c1:=0                                 c2:=0
          end                                   end
      end;                                  end;
    /* critical section 1 */              /* critical section 2 */
    c1:=1;                                c2:=1;
    turn:=2                               turn:=1
  end                                   end
```

Fig. 2. Dekker's mutual exclusion solution

```
4:   [P2(1) : c2:=1]
5:   <P2(12) : true> yes
6: <P1(12) : true> yes
7: [P1(2) : c1:=0]
8:   [P2(2) : c2:=0]
9: <P1(8) : c2=0?> yes
10:  <P2(8) : c1=0?> yes
11:<P1(7) : turn=2?> no
12:  <P2(7) : turn=1?> yes
13:  [P2(3) : c2:=1]
14:<P1(8) : c2=0?> yes
15:[P1(9) : /* critical-1 */]
```

A different execution σ_2 can be obtained with the same initial state. Process $P2$ sets $c2$ to 0 (line 7), signaling that it wants to enter its critical section. It is faster than process $P1$, and manages also to check whether $c1$ is 0 (line 8) before $P1$ changes it from 1. Hence $P2$ enters its critical section (line 9).

```
1:(P1(0) : start)
```

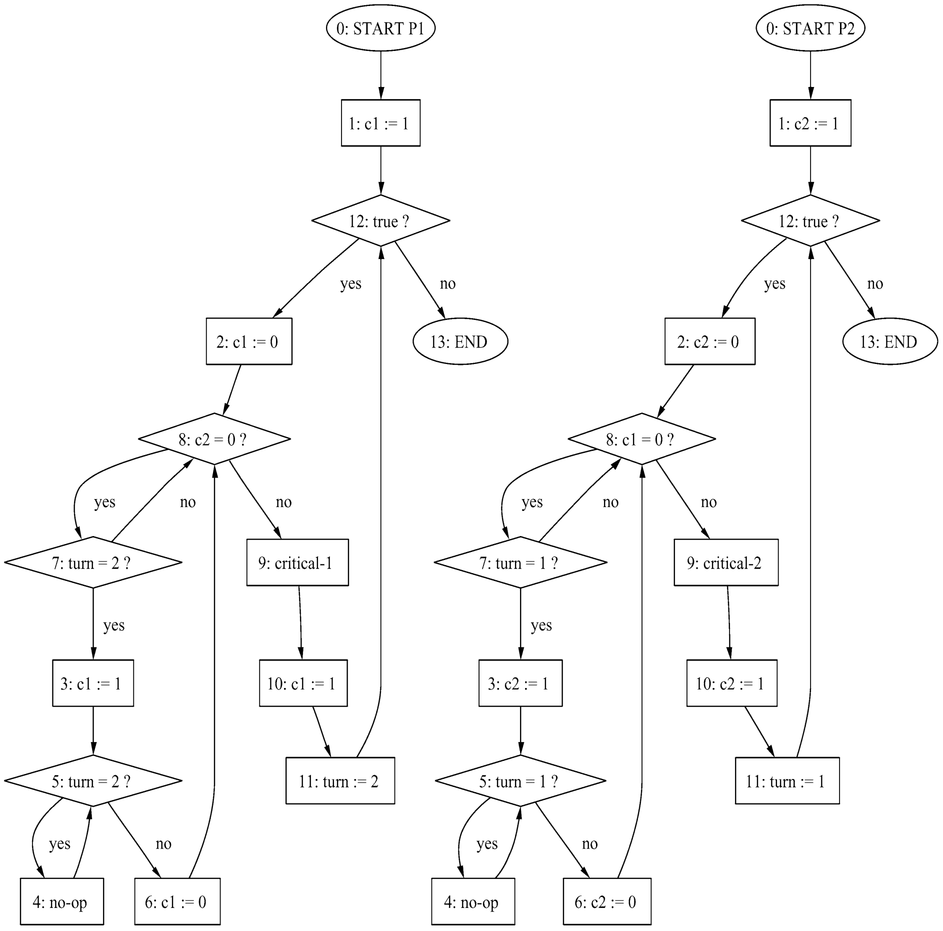

Fig. 3. Dekker's mutual exclusion solution

```
2:  (P2(0) : start)
3:[P1(1) : c1:=1]
4:  [P2(1) : c2:=1]
5:  <P2(12) : true> yes
6:<P1(12) : true> yes
7:  [P2(2) : c2:=0]
8:  <P2(8) : c1=0?> no
9:  <P2(9) : /* critical-2 */>
```

A (global) *state* of a program is a function that assigns values to the program variables, including the program counters. We assume that the program can be translated into a set of atomic actions A. Each atomic action consists of a *condition* and a *multiple assignment* (changing the values of some program variables, including program counters). Some of the conditions are implicit to the text of the program, e.g., a check that a program counter has a particular value. Similarly, part of the multiple assignment does not come directly from the code of the program; in particular each action includes a change to a program counter variable. For example, a transition for node 3 in process $P1$ in Figure 3 can have a condition $pc1 = 3$ and a multiple assignment $(pc1, c1) := (5, 1)$. That is, $pc1$ obtains the value 5 and $c1$ the value 1. Here $pc1$ is the program counter for process $P1$. The actions are partitioned between the different processes P. We denote by $A(p_i)$ the action of A that belongs to the process $p_i \in P$.

An *execution sequence* (or *behavior*) σ is an alternating sequence of states and actions $s_1, \alpha_1, s_2, \alpha_2 \ldots \alpha_{n-1} s_n$, where each state s_{i+1}, for $1 \leq i < n$, is obtained from its predecessor s_i by executing the action α_i. This means that the condition of the action α_i holds in s_i and the multiple assignment associated with α_i is applied to s_i, resulting in the state s_{i+1}. Given a fixed initial state s_1, and the fact that actions are deterministic (note that this does not mean that the code is determinitic, as multiple actions may be enabled at the same state), we can equivalently represent an execution as then pair $\langle s_1, \alpha_1\alpha_2 \ldots \alpha_{n-1}\rangle$.

The *dependency relation* $D \subseteq A \times A$ is a reflexive and symmetric relation over the actions. It captures the cases where concurrent execution of actions cannot exist. Thus, $(\alpha, \beta) \in D$ when

- α and β are in the same process, or
- α and β use or define (update) a mutual variable[1] .

This dependency corresponds to the execution model of concurrent programs with shared variables, as used in Section 3. A different dependency relation can be defined for other models of concurrency, see e.g., Section 5. Let σ be a sequence of actions from A. We can index the kth time that α appears on σ with k, obtaining α_k. We call α_k the kth *occurrence* of α in σ. Thus, instead of the sequence of actions $\alpha\alpha\beta\alpha\beta$, we can equivalently denote σ as the sequence of occurrences $\alpha_1\alpha_2\beta_1\alpha_3\beta_2$. We denote the set of occurrences of a sequence σ by E_σ. In the above example, $E_\sigma = \{\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2\}$.

Define a binary relation $\rightarrow_\sigma$ between occurrences on a sequence σ. Let $\alpha_k \rightarrow_\sigma \beta_l$ on a sequence σ when the following conditions hold:

(i) α_k occurs before β_l on σ.

(ii) $(\alpha, \beta) \in D$.

Thus, $\alpha_k \rightarrow_\sigma \beta_l$ implies that α and β refer to the same variable or belong to the same process. Because of that, α_k and β_l cannot be executed concurrently. According to the sequence σ, the imposed order is α_k before β_l. Let $\rightarrow_\sigma^*$ be the transitive closure completion of $\rightarrow_\sigma$. This is a partial order, since it is transitive, asymmetic and irreflexive. The *partial order view* of an execution σ is $\langle E_\sigma, \rightarrow_\sigma^*\rangle$.

We want to impose synchronization of the checked code according to the

[1] depending on the hardware, we may allow α and β to be concurrent even if both only use a mutual variable but none of them define it.

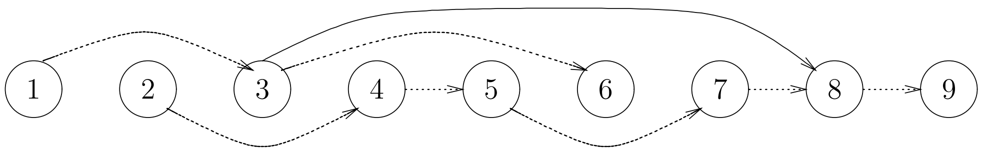

Fig. 4. The order between occurrences in the latter execution

relation $\rightarrow_\sigma$. However, this relation contains many pairs, which means a lot of synchronization. This can be inefficient to implement and induce many changes from the original code. We can reduce $\rightarrow_\sigma$ by removing pairs of actions $\alpha_k \rightarrow_\sigma \beta_l$ that belong to the same process or that have a chain of related (according to $\rightarrow_\sigma$) occurrences between them. The *reduced* relation between occurrences of σ is denoted by $\leadsto_\sigma$. It is defined to be the (unique) relation satisfying the following conditions:

(i) The transitive closure of $\leadsto_\sigma$ is $\rightarrow_\sigma$.

(ii) There are no elements α_k, β_l and γ_m such that $\alpha_k \leadsto_\sigma \beta_l$, $\beta_l \leadsto_\sigma \gamma_m$ and $\alpha_k \leadsto_\sigma \gamma_m$.

Due to the physical nature of concurrent programs, it is sufficient to maintain the synchronization according to the $\leadsto_\sigma$ order. The rest of the orders according to $\rightarrow_\sigma$ are guaranteed by transitivity and by the sequentiality of the individual processes.

Calculating the relation $\leadsto_\sigma$ from $\rightarrow_\sigma$ is simple. We can adapt the Floyd-Warshall algorithm [6,25] for calculating the transitive closure of $\rightarrow_\sigma$. Each time a new edge is discovered as a combination of existing edges (even if this edge already exists in $\rightarrow_\sigma$), it is marked to be *removed.* At the end, we remove all marked edges.

Figure 4 consists of the graph of occurrences that correspond to the execution sequence σ_2 earlier in this section. The nodes in this figure are labeled according to the sequence numbers listed in σ_2. The arrow from node 3 to 8 corresponds to the update and use of the same variable ($c1$) by the different processes (and according to $\leadsto_\sigma$), while the rest of the arrows correspond to ordering between actions belonging to the same process.

Although an execution is represented by a sequence, we are in general interested in a *collection* of *equivalent* executions. To define the equivalence between executions, let $\sigma|_{\alpha,\beta}$ be the projection of the sequence σ that keeps only occurrences of α and β. Then $\sigma \equiv_D \rho$ when $\sigma|_{\alpha,\beta} = \rho|_{\alpha,\beta}$ for each pair of interdependent actions α and β, i.e., when $(\alpha, \beta) \in D$. This also includes the case where $\alpha = \beta$, since D is reflexive. This equivalence is also called *partial order* equivalence or *trace* equivalence [14]. It relates sequences σ and ρ for the following reasons:

- The same occurrences appear in both σ and ρ.
- Occurrences of actions of a single process are interdependent (all the ac-

tions of a single process use and define the same program counter). Thus, each process executes according to both σ and ρ the same actions in the same order. This represents the fact that processes are sequential.

- Any pair of dependent actions from different processes cannot be executed concurrently, and must be sequenced. Their relative order is the same according to both σ and ρ.
- Occurrences of independent actions that are not separated from each other by some sequence of interdependent actions can be executed concurrently. They may appear in different orders in trace-equivalent executions. In enforcing an execution, we do not want to impose new synchronizations that will order actions that can be executed concurrently. Another way to look at this is that distinguishing between two equivalent executions can only be done by having global clocks and making experiments that are expensive and unnatural to concurrent systems [12].

The partial order view was studied extensively as an alternative for the more traditional interleaving (linear) semantics [17,21]. The advantage of the interleaving semantics is that it can be handled with simpler mathematical tools (such as linear temporal logic [20] or Büchi automata [23]) than the tools needed for partial order semantics. On the other hand, partial order semantics gives a more intuitive representation of concurrency. The recent interest in partial order representation of concurrent systems stems from the popularity of the ITU standard for message sequence charts [11]. We propose that the rationale to recover the concurrent execution related to the behavior, rather than a completely synchronous linear execution, gives another motivation for using the partial order semantics.

We define several operators on execution sequences, represented as sequences of actions (hence, in this case, ignoring the states):

$\mathit{Hide}_B(\sigma)$ The sequence σ after removing (projecting out) the actions from the set B.

$\mathit{Cl}_D(\sigma)$ The set of sequences obtained from σ by making repeated permutations between adjacent actions that are independent, i.e., not related by D. That is, $\mathit{Cl}_D(\sigma) = \{\rho | \rho \equiv_D \sigma\}$.

$\mathit{Lin}(E, <)$ The set of linearizations (completions to total orders) of the partial order $\langle E, <\rangle$.

$\mathit{Exec}(P)$ The set of executions of a program P.

The operators defined here over sequences can be easily extended to sets of sequences, e.g.,

$$\mathit{Hide}_B(S) = \bigcup_{\sigma \in S} \mathit{Hide}_B(\sigma)$$

Lemma 2.1 *We have the following relation [14], connecting the above:*

(1) $$\mathit{Cl}_D(\sigma) = \mathit{Lin}(E_\sigma, \rightarrow_\sigma^*)$$

That is, the trace-equivalent sequences obtained by shuffling independent events in σ are the linearizations of the partial order view of σ.

This means that the partial order view and the trace equivalence are dual ways of looking at the same thing. This will help us to formalize the outcome of our program transformation.

In order to also take into account the preservation of the temporal properties, we present an additional view of an execution σ: Let $\mathcal{P}$ be a set of propositions in some given temporal specification (e.g., as a linear temporal logic specification or using an automaton). Let $T : \mathcal{P} \mapsto \{\mathit{true}, \mathit{false}\}$ be a truth assignment over it. A *propositional sequence* is a (finite or infinite) sequence of truth assignments over some given set of propositions. Interpreting the propositional variables in each state of a sequence σ (this time ignoring the actions) results in a *propositional sequence*.

3 Transforming Shared Variables Programs

We first assume a computational model of several concurrent processes with shared variables. Each process is coded in some sequential programming language such as C or Pascal. Although no explicit nondeterministic choice construct exists, an overall nondeterministic behavior of the program can be the result of the fact that the processes can operate in different relative speeds. Hence, even if we start the execution of the code with exactly the same initial state, we may encounter different behaviors as the sequences σ_1 and σ_2 from Section 2 demonstrate. Our goal is then to enforce, under the given initial condition, that the program executes in accordance with the suspicious behavior.

In order to perform the transformation, we prepare the code, while translating it into a set of actions. We keep pointers to the text location corresponding to the beginning and end of actions. In most cases, the transformation consists of adding code at these locations, i.e., before or after an action. For simplicity, we start presenting the transformation with the unrealistic assumption that we can add code for the existing actions in a way that maintains the atomicity of the actions. Since this will result in rather large actions, which cannot realistically be executed atomically, we split them in a way that is detailed and explained below.

For each pair of processes p_i and p_j, $p_i \neq p_j$, such that for some occurrences α_k with $\alpha \in A(p_i)$, and β_l with $\beta \in A(p_j)$, $\alpha_k \rightsquigarrow_\sigma \beta_l$, we define a variable V_{ij}, initialized to 0. It is used by process p_i to inform process p_j that it can progress. This is done in the style of the usual *semaphore* operations [5] (it can be proved that a binary semaphore is sufficient here). The process p_i does that by incrementing V_{ij} after executing α_k.

$$\text{Free}_{ij} : V_{ij} := V_{ij} + 1$$

The process p_j waits for the value of this variable to be 1 and then decrements it.

$$\text{Wait}_{ji} : \mathit{wait}\, V_{ij} > 0; V_{ij} := V_{ij} - 1$$

Let $S(p_i) \subseteq A(p_i)$ be the set of actions of process p_i that have an occurrence in σ and are related by $\rightsquigarrow_\sigma$ to an occurrence of an action in another process. Thus, $S(p_i)$ are the actions that have some (but not necessarily all) occurrences that need to be synchronized. Thus, we need to check whether we are currently executing an occurrence of an action $\alpha \in S(P_i)$ that requires synchronization. Let $count_i$ be a new local counter variable for process p_i. We increment $count_i$ *before* each time an action from $S(p_i)$ occurs, i.e., add the code

$$count_i := count_i + 1 \tag{2}$$

immediately after the code for α.

We define $\sharp_i \alpha_k$ to be the number of occurrences from $S(p_i)$ that appeared in σ before α_k. We can easily calculate $\sharp_i \alpha_k$ according to the sequence σ. This is also the the value that the variable $count_i$ has during the execution of the code after the transformation, due to the increment statement in (2).

Suppose now $\alpha_k \rightsquigarrow_\sigma \beta_l$, where $\alpha \in A(p_i)$, $\beta \in A(p_j)$, $p_j \neq p_j$. Then we add the following code *after* α_k:

$$\mathit{if}\ count_i = \sharp_i \alpha_k\ \mathit{then}\, \text{Free}_{ij} \tag{3}$$

We add the following code *before* β_l:

$$\mathit{if}\ count_j = \sharp_j \beta_l\ \mathit{then}\, \text{Wait}_{ji} \tag{4}$$

The notations $\sharp_i \alpha_k$ and $\sharp_j \beta_l$ should be replaced by the appropriate constants calculated from σ. Since an action may participate in several occurrences on the same sequence, different code akin to (3) and (4) for multiple occurrences can be added. We can optimize the transformation by not counting (and not checking for the value of $count_i$ in) actions that appear only once in the given execution σ. Similarly, we do not need to count actions that require the same added transformation code in all their occurrences.

Another consideration is to identify when the execution is finished. We can add to $S(p_i)$ the action α that appears last in the execution per each process p_i. Thus, we count the occurrences of α as well in $count_i$. Let $\sharp_i \alpha_k$ be the value of $count_i$ for this last occurrence α_k of α in σ. We add the following code, after the code for α:

$$\mathit{if}\ count_i = \sharp_i \alpha_k\ \mathit{then}\ \mathit{halt}\, p_i \tag{5}$$

(there is no *halt* statement in Pascal, so it is implemented using a *goto*.) Again, if the last action of the process is the only occurrence of α, we do not need to count it. Note that if we do not halt the execution of the process p_i here, we may encounter and perform a later action of p_i that is not in σ and is dependent of an action of another process that did not reach its last occurrence in σ. This may lead to a behavior quite different than the one we investigate.

In order to minimize the effect of the additional code on executions other than the suspicious execution (or those equivalent to it, under $\equiv_D$), we use an additional flag $check_i$ (for each process p_i), whose value remains constant throughout the execution. This flag is *true* only when we run the code in the mode where we want to repeat the suspicious behavior. Thus, in addition to the distinguished initial state for the execution, we also force $check_i = true$ for each process p_i. In all other cases, we set initially $check_i = false$. When $check_i = false$, even if we start the execution according to the initial state of the suspicious behavior, the program may follow an execution different than the suspicious one. Note that we chose not to have one global variable *check*, since the different references of it by different processes would be interdependent and hence would defy our goal to preserve the concurrent structure of the execution.

Now, if *Code* is some code generated by our transformation, as described in (2)–(5), we wrap it with a check that we are currently tracing a given sequence, as follows:

$$if\ check_i\ then\ Code$$

Some code simplification may be in place, for example, checking the value of $count_i$ and the value of $check_i$ can be combined to a single *if* statement.

As stated above, modeling the additional code resulted from the transformation as amalgamated into the atomic actions of the original code is unrealistic. The behavior of the resulted code better corresponds to adding new actions. However, we have carefully constructed it so that it comprises of additional actions that are mostly local to a process, i.e., independent of actions of other processes. The only additional dependent actions are of the form $Free_{ij}$ and $Wait_{ji}$. However, when such a pair is added, $Free_{ij}$ would be preceded by some action α_k, and $Wait_{ji}$ is succeeded by an action β_l such that $\alpha_k \rightarrow_\sigma \beta_l$. The dependence of these actions ($Free_{ij}$ and $Wait_{ji}$) are the same as the existing ones (α_k and β_l). Moreover, it can be shown that there cannot be any occurrence of an action between α_k or $Free_{ij}$ (both in $A(p_i)$) that are dependent on actions from p_j. Similarly, all occurrences between the occurrence of $Wait_{ji}$ and β_l (both from p_j) are independent of actions from p_i. Hence the concurrency structure of the program is maintained, even when we break the actions of the transformed program in a more realistic way. We can formalize this issue as action refinement [24] under the partial order semantics. However, we prefer to keep the presentation on the intuitive level rather than on the theoretical-formal one.

Advanced note. In order to avoid introducing unnecessary delays or even deadlocks, we must guarantee that the implementation of Wait_{ji} must not block the process p_i. Specifically, if V_{ij} is 0 and process p_j is waiting for it to become 1, process p_i needs to be able to progress, which will allow it to eventually increment V_{ij}. Such blocking could exist, e.g., on a single processor multitasking the concurrent program, with $Wait_{ji}$ performing busy waiting for V_{ij} to become 1, the scheduler is unfair to process p_i. It is interesting that

```
boolean c1, c2, check1, check2;
boolean V12 initially 0;
integer (1..2) turn;

P1::c1:=1;
  if check1 then V12:=1;
  while true do
  begin
    if check1 then halt P1;
    c1:=0;
    while c2=0 do
      begin
        if turn=2 then
          begin
            c1:=1;
            while turn=2 do
             begin
              /* no-op */
             end;
            c1:=0
          end
      end;
    /* critical section 1 */
    c1:=1;
    turn:=2
  end

P2::c2:=1;
  while true do
  begin
    c2:=0;
    if check2 then
      begin wait V12>0;
       V12:=0 end
    while c1=0 do
      begin
        if turn=1 then
          begin
            c2:=1;
            while turn=1 do
             begin
              /* no-op */
             end;
            c2:=0
          end
      end;
    /* critical section 2 */
    if check2 then halt P2;
    c2:=1;
    turn:=1
  end
```

Fig. 5. The transformed Dekker algorithm

the sequence σ_3 in Section 6 shows a situation in the Dekker algorithm that is related to such a case. (Hence, we will demonstrate, using our running example, a subtle concurrency problem in the Dekker algorithm, which we need to avoid in the implementation of our transformation.)

Lemma 3.1 *If no modeling errors were made, and the actions in the given suspicious execution correctly follow the original code, then our construction does not generate new deadlocks.*

The transformed Dekker algorithm, which allows checking the path σ_2 in section 2 is shown in Figure 5. The added code appears in boldface letters.

In order to reason about the program transformations we propose, we will denote the original program actions by A and the augmented set of actions by $A' \supset A$ (some minor changes can be inflicted on the actions A, in particular, changes to program counter values, albeit there is no change in the code corresponding to these actions). We denote the dependency between the program actions A by the symmetric and reflexive relation $D \subseteq A \times A$. Adding new actions $A' \setminus A$ results in a new dependency relation $D' \subseteq A' \times A'$. We have that $D' \cap (A \times A) = D$, i.e., the program transformations do not add any dependencies between the original transitions.

Lemma 3.2 *Let P be the original program, and P' be the result of the transformation. Then we can be expressed as follows:*

$$Hide_{A' \setminus A}(Exec(P')) = Cl_D(\sigma) \tag{6}$$

That is, when hiding the additional actions from the executions of the transformed program, we can obtain any execution that is equivalent under $\equiv_D$ to the sequence σ.

4 Preserving the Checked Property

Our transformation so far enables us to control the execution of the program in such a way that we are restricted to executions that are trace-equivalent to the counterexample. Suppose further that the execution σ was obtained using a model checker, which was used to verify whether some concurrent program satisfies a property φ. We abstractly assume that the property φ corresponds to a set of propositional sequences. In practice it can be specified e.g., using a temporal formula or an automaton over (finite or infinite) sequences. Since σ is a counterexample, it typically satisfies $\neg\varphi$.

Denoting $L(\varphi)$ as a set of propositional executions (or a sequence of actions, depending on the type of specification), the difficulty appears when φ is not closed under the trace equivalence [19]. That is, we can have $\rho \equiv_D \sigma$ where $\rho \in L(\varphi)$ and $\sigma \in L(\neg\varphi)$. There are several solutions for this situation. One is to use a specification formalism that is closed under trace equivalence (see e.g., [1,10,22]). Another solution is to use a specification formalism that does not force trace closedness, and then apply an algorithm for checking whether φ is closed. Such an algorithm appears in [19].

We propose here a third possibility, where we do not enforce φ to be trace-closed. The idea is to add dependencies so that the trace equivalence is refined, and equivalence sequences do not differ on satisfying φ. We construct a graph $G = \langle \sigma, S, \Rightarrow \rangle$. Each node in S represents an execution sequence from $Cl_D(\sigma)$. The *initial* node is $\sigma \in S$. An edge $\rho \Rightarrow \rho'$ exists if ρ' is obtained from ρ using the switching of a single adjacent pair of independent actions. Starting the search from σ, which satisfies φ, we check each successor node for the satisfaction of φ. Given that ρ satisfies φ but its successor ρ' satisfies $\neg\varphi$, we add synchronization that prevents the permutation of ρ into ρ'. That is, if $\rho = \mu\alpha_k\beta_l\mu'$ and $\rho' = \mu\beta_l\alpha_k\mu'$ for some prefix μ and suffix μ', and occurrences β_l and α_k, we add a synchronization $\alpha_k \rightarrow_\sigma \beta_l$. Note that for optimization, we did not make the *actions* α and β interdependent, but rather synchronized two specific *occurrences*. We can reduce $\rightarrow_\sigma$ using the adaptation of the Floyd-Warshall algorithm, as presented in Section 2.

This algorithm provides a small number of additional synchronizations for preserving the temporal property. However, its complexity is high. The size of $Cl_D(\sigma)$ may be at worst exponential in the length of σ. Other heuristic solutions are possible. For example, we can follow the partial order reduction strategies (ample sets, persistent sets or stubborn sets) and check which actions may change propositions participating in φ. Then we make all such actions interdependent. This solution is good for a specification that is stuttering closed. Details and further references can be found e.g., in Chapter 10 of [3]. This solution has a much better complexity (quadratic in the number of actions), but is suboptimal since it may add some redundant synchronizations.

5 The Distributed Programs Model

Consider now a different distributed systems model, where we have a handshake (synchronous) message passing instead of shared variables. Other models, such as buffered communication can be handled in a similar way, following the description in this and the previous section. We assume that our program has the following kinds of actions:

- Local actions, related to a single process.
- Communication actions. We assume here a handshake communication, as in Ada or CSP. Such an action is executed jointly (and simultaneously) by a pair of processes.

We can again assign dependencies to the actions. We have that $(\alpha, \beta) \in D$ when α and β participate in a mutual process. Note that in particular a communication participates in a pair of processes, hence it depends on actions from both processes.

We use the following syntactic construct:

$$\textit{select}\ S_1 [] S_2 [] \ldots [] S_n\ \textit{end}$$

where the code for S_i starts with a communication (*send* or *receive* a message), after a potential local condition, i.e, a 'guard'. One syntax for *guarded* communication [8] is of the form $g \Rightarrow c$, where g is a local condition, and c is a communication. In turn, c can be of the form $P!e$, where P is a process and e is an expression whose calculated value is sent to P, or $Q?x$, where Q is a process, and x is a variable to receive the sent value. The joint effect of $P!e$ on a process Q and $Q?x$ on process P is as if $x := e$ was executed by the two processes, P and Q.

The *select* itself is not translated into an action. It is only a keyword that allows several communication actions to be potentially enabled at that location.

We add again a local counter $count_i$ for each process. The counter is incremented before each communication action inside a select. We can thus check if the value of $count_i$ is $\sharp_i\alpha_k$ in order to select according to the given execution. We can then replace the *select* statement with a deterministic code that chooses the appropriate communication according to the suspicious execution. For example, consider the select statement

$$\text{select } \beta \; [] \; \gamma \text{ end}$$

Suppose that β (together with a matching communication from another process) occurs in σ as β_j and β_k and γ occurs as γ_l and γ_m. We replace the *select* statement with the following code:

```
case $count_i$ of
            $\sharp_i\beta_j$, $\sharp_i\beta_k$ : $\beta$;
            $\sharp_i\gamma_l$, $\sharp_i\gamma_m$ : $\gamma$
end
```

Thus, if $count_i$ is either $\sharp_i\beta_j$ or $\sharp_i\beta_k$ we need to choose the communication action β. In the other two cases, we need to choose γ. Note that since a communication is shared between two processes, it would be counted separately by both.

As in the case of programs with shared variables, we need to add code for activating the additional checks only when enforcing a suspicious execution, i.e., when $check_i = true$. Similarly, we include the last action in every process in the counting, in order to halt the execution. Note that with no shared variables and under handshake communication, the code added in the transformation is completely local to the processes.

Lemma 5.1 *If no modeling errors were made, and the actions in the given suspicious execution correctly follow the original code then the transformation does not introduce new deadlocks.*

Note that if we want to enforce preserving the checked property on all executions that are trace equivalent to the suspicious one (the execution we want to monitor), we can apply the transformation given in Section 4. This means adding semaphores and, consequently, shared variables, even when the original code includes only interprocess communication.

6 Extending the Framework to Infinite Traces

A model checker may generate an infinite execution that fails to satisfy the given specification. Although infinite, such a sequence is *ultimately periodic* [23]. It consists of a finite *prefix* σ and a finite recurrent sequence ρ. This is often denoted as $\sigma\rho^\omega$. We can apply our transformation with some small changes to the two finite parts, σ and ρ. Of course we cannot execute $\sigma\rho^\omega$, since it is infinite, but we can test its execution for any given finite length (depending on our patience).

The first change is to adapt the counting of the actions involved in the synchronization, and in the last action per process to behave differently according to σ and ρ. We add a variable $phase_i$ for each process p_i, initialized to 0. We do not halt the execution with σ. Instead, when we reach the last action of process p_i in σ (as described in Section 3), we update $phase_i$ to 1, and behave according to the execution ρ. When we reach the end of σ, and each time we reach the last current process action according to ρ, we reset $count_i$ to zero.

Let $G(\rho) = \langle P, E\rangle$ be an undirected graph, whose nodes are the processes, and an edge between p_i and p_j exists if there are occurrences α_k, β_l of ρ such that $\alpha_k \rightsquigarrow_\sigma \beta_l$ or $\beta_l \rightsquigarrow_\sigma \alpha_k$, $\alpha \in A(p_i)$, $\beta \in A(p_j)$.

There are three cases for the ultimately periodic part ρ that can be distinguished:

(i) The graph $G(\rho)$ includes all the processes in one connected component (a maximal component of nodes such that there exists a path between each pair of nodes in the component). In this case, in the enforced execution, some occurrence of the ith iteration of ρ may be overtaken by the $i+1$st iteration of ρ, due to concurrency (independence). However, such overtaking is limited, and events from the $i+2$nd iteration cannot overtake any event in the ith iteration.

(ii) The graph $G(\rho)$ consists of multiple disjoint connected non-singleton components. In this case, the behavior is as if the components iterate independently, and there can be an unbounded overtaking between them[2]. A similar behavior is obtained when concatenating message sequence charts [16].

(iii) The graph $G(\rho)$ includes only singleton components (i.e., consisting of a single nodes each). In this case, it is possible that some processes were not given a fair chance to continue. Then, the construction is not guaranteed to behave according to $\sigma\,\rho^\omega$. Running the transformed program may result in additional actions from the underrepresented processes to be executed. Because of shared variables or message passing, this can affect the processes that are represented. Because our transformation inserts some code in which a process may wait for another *based on the suspicious execution*, our transformation may result in a deadlock. In fact, such a discrepancy between the execution found during the program analysis (verification, testing) and the actual behavior is a useful information. It indicates that the analysis did not take into account some important semantic consideration (fairness), and therefore increases the possibility that the given execution is a false negative.

We provide here an example for an ultimately periodic sequence σ_3, which

[2] This distinction is related to the definition of the concurrent star operator $c*$ [4] and its related infinite version $c\omega$. The suspicious behavior is defined to be $\sigma\rho^\omega$, but without explicitly synchronizing each repetition of ρ, we actually achieve $\sigma\rho^{c\omega}$.

indicates that a livelock occurs. Process $P1$ is occupied in an infinite loop, waiting for process $P1$ to relinquish its attempt to get into the critical section, while process $P2$ is making no progress. The finite prefix of σ_3 is as follows:

```
1: (P1(0) : start)
2:   (P2(0) : start)
3: [P1(1) : c1:=1]
4:   [P2(1) : c2:=1]
5:   <P2(12) : true> yes
6: <P1(12) : true> yes
7: [P1(2) : c1:=0]
8:   [P2(2) : c2:=0]
9: <P1(8) : c2=0?> yes
10:  <P2(8) : c1=0?> yes
11:<P1(7) : turn=2?> no
12:  <P2(7) : turn=1?> yes
13:  [P2(3) : c2:=1]
```

The recurrent part of σ_3 consists of the following two occurrences:

```
14:  <P2(5) : turn=1?> yes
15:  [P2(4) : /* no-op */]
```

The initial state is the same as in the previous example executions. In the ultimately periodic part, process $P1$ is not contributing to the execution, while $P2$ loops, waiting for $turn$ to become 2 (lines 14 and 15). Since $P1$ does not execute, $turn$ remains 1 and $P2$ never goes out of its loop. This execution can be the result of an analysis that does not take fairness into account. With fairness, process $P1$ would continue, and will check the value of $c2$, which now becomes 1; hence $P1$ can continue into its critical section, and eventually set $turn$ to 2. Consequently, $P2$ will eventually be able to get into its critical section.

7 Implementation and Discussion

The suggested algorithm was implemented by the second author. It can be used independently or as an extension to the PET system [7]. The algorithm takes as input a concurrent Pascal program and outputs the transformed program. We used the PET system to test the implementation in the following way. PET is designed to check concurrent executions. Given a manually selected sequence of flow graph nodes, corresponding to the given program (selected by clicking a sequence of nodes from flow graphs that are automatically generated by PET from the given Pascal processes), it calculates their path condition. As discussed in Section 2, because of nondeterminism, such a condition is not sufficient to force the selected sequence. Thus, the algorithms presented here are exactly the extension needed for forcing the selected path (up to trace equivalence).

We can now feed the transformed code, including an assignment that forces the given initial condition. With that assignment, the new path condition for each one of the sequences that are trace equivalent to the originally selected path is *true*. PET also allows permuting pairs of adjacent independent events by selecting the first of them and clicking the mouse. We can also select other sequences, not equivalent to the original one.

We presented a program transformation that forces a program to execute according to a given scenario. The changes to the program code inspired by the transformation are minimal, allowing it to also have the other executions, when not started from a particular given initial state (in which, in particular, $check_i = true$ for each process p_i). This transformation can be used to check whether a suspicious behavior reported by a model checker or a theorem proving effort is indeed faulty. The transformation can also be used by testers, who need to demonstrate that a discovered error actually occurs. Due to the highly nondeterministic nature of concurrent programs, it may be difficult to enforce a suspicious scenario, which may depend on an infrequent scheduling choices. The transformation preserves the concurrent structure of the program. Such a simple transformation can be verified or comprehensively tested in order to gain confidence in its correctness. Thereafter, it can be used as a standard tool for testing the results of verification tools.

References

[1] Alur R., Peled D., W. Penczek (1995), Model-Checking of Causality Properties, LICS'95, 10th Symposium on Logic in Computer Science, IEEE, 1995, 90–100,

[2] H. Ben-Abdullah, S. Leue, Syntactic detection of process divergence and non-local choice in Message Sequence Charts, TACAS 1997, LNCS 1217, Springer, 259–274.

[3] E. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press 1999.

[4] V. Diekert, G. Rozenberg, The Book of Traces, World Scientific, 1995.

[5] E. W. Dijkstra, The structure of the THE multiprogramming system, Communication of the ACM 9(1976), 27–37.

[6] R.W. Floyd. Algorithm 97 (Shortest Path). *Communications of the ACM* **5** (1962), pp. 365.

[7] E.L. Gunter, D. Peled, Temporal Debugging for Concurrent Systems, TACAS 2002, Grenoble, France, LNCS 2280, Springer, 431-444.

[8] C.A.R. Hoare, Communication Sequential Processes, Prentice-Hall, 1985.

[9] J. Grabowski, B. Koch, M. Schmitt, D. Hogrefe, SDL and MSC based test generation for distributed test architecture, SDL 99 The Next Millenium, Elsevier, June 1999.

[10] S. Katz and D. Peled. Interleaving set temporal logic. *Theoretical Computer Science* 75, 21–43, 1992.

[11] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), March 1993.

[12] L. Lamport, Time clocks, and the ordering of events in a distributed system, CACM 21(1978), 558–565.

[13] T. J. LeBlanc, J. M. Mellor-Crummey, Debugging Parallel Programs with Instant Replay, IEEE Transactions on Computers, C-36 (4), 471–482, 1987.

[14] A. Mazurkiewicz, Trace Theory, *Advances in Petri Nets* 1986, Lecture Notes in Computer Science 255, Springer-Verlag, 1987, 279–324.

[15] B. Mitchell, Coordinating parallel test components in distributed tests,

[16] A. Muscholl, D. Peled, Z. Su, Deciding Properties for Message Sequence Charts, *FoSSaCS, Foundations of Software Science and Computation Structures*, Lisbon, Portugal, 1978, LNCS 1378, 226-242.

[17] M. Nielsen, G. Plotkin, G. Winskel, Petri Nets, Event Structures and Domains, Part I, Theoretical Computer Science 13(1981), 85–108.

[18] S. Owicki, D. Gries, Verifying properties of parallel programs: an axiomatic approach, CACM 19(1976), 279–285.

[19] D. Peled, Th. Wilke, P. Wolper, An algorithm approach for checking closure properties of temporal logic specifications and omega-regular languages, TCS 195(1998), 183–203.

[20] A. Pnueli, The temporal logic of programs, 18th IEEE symposium on Foundation of Computer Science, 1977, 46–57.

[21] V. Pratt, Modeling concurrency with partial orders, International Journal of Parallel Programming, 15 (1986), 33–71.

[22] P.S. Thiagarajan, I. Walukiewicz, An expressively complete linear time temporal logic for Mazurkiewicz traces, LICS'97, IEEE, 1997, 183–194.

[23] W. Thomas, Automata on infinite objects, In Handbook of Theoretical Computer Science, vol. B, J. van Leeuwen, ed., Elsevier, Amsterdam (1990) 133–191.

[24] W. Vogler, Bisimulation and action refinement, Theoretical Computer Science 114(1993), 173–200.

[25] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, **9** (1962), pp. 11-12.