7 Shared Variables Concurrency

7.1 A Characterisation of Concurrent Execution

In this section a model is developed for characterising the influence upon the program state of the execution of concurrent programs P that operate upon shared variables. Such a program $P \equiv P_1 \| \ldots \| P_n$ consists of one or more processes $P_1, \ldots, P_n$ which communicate with each other by reading and updating the values of so-called *shared* variables. Shared variables of $P_1, \ldots, P_n$ are variables which are accessed by two or more processes – whence the name "shared"; the remaining variables of $P_1, \ldots, P_n$ are called *local*, because they are accessed by merely one process P_i, $i = 1, \ldots, n$.

During program execution several processes may *compete* in accessing the same shared variable. Since ultimately such a shared variable models a particular memory location, and in current computers more than one process cannot simultaneously access the same memory location, the outcome of such a competition is that accesses to the same shared variable are linearly ordered, or interleaved, as the lingo goes. This explains why two enabled actions, whose execution requires them to access the same shared variable, are said to be in *conflict*. Another aspect of concurrency is *independent* simultaneous execution whenever the shared variables, which are accessed, are all different. Besides these two elements – *competition* for access to the same variable, and *independent* operation on disjoint sets of variables – a third element which characterises execution of concurrent programs in this chapter is that each process involved has a positive, although arbitrary, speed of execution, unless it is terminated, deadlocked or waiting to obtain access to a shared variable, in which case we postulate that access is eventually granted after waiting long enough. By convention this also includes accesses to local variables, implying that every enabled action operating on purely local variables will be eventually executed.

In a first approximation, the execution of a concurrent program therefore satisfies the following two requirements:

Requirement 1: More than one process cannot have simultaneous access to the same shared variable.

Requirement 2: The execution speed of every nonterminated and nondeadlocked process is positive and arbitrary, unless it is waiting for access to a shared variable, in which case by waiting long enough access is eventually granted and the process resumes execution.

Next we introduce the notion of an *atomic action* using a purely textual representation of programs in case their representation as a transition system is unambiguous. This is the case for, e.g, $R_1 :\ x := 1$ and $R_2 :\ x := 2$. What is the effect of executing $R_1 \| R_2$? By **Requirement 1**, R_1 and R_2 cannot access

x simultaneously. However, the result of $R_1 \| R_2$ is by no means fixed, for this depends, as we shall see below, upon the way $x := 1$ and $x := 2$ are executed. First consider the following two-bit implementation of these actions:

$$\begin{array}{l} R_1 : \ x := 1 \Longleftrightarrow \textit{first bit of}\ (x) := 1; \textit{second bit of}\ (x) := 0, \ \text{and} \\ R_2 : \ x := 2 \Longleftrightarrow \textit{first bit of}\ (x) := 0; \ \textit{second bit of}\ (x) := 1, \end{array}$$

where, for $i = 0, 1$, *first bit of* $(x) := i$ and *second bit of* $(x) := i$ are regarded as single uninterruptable actions. Such uninterruptable actions are called atomic, because during their execution no interference by other actions from other processes takes place together with their execution. That is, during execution of an atomic action the set of objects manipulated by that action (i.e., whose values are read or changed) and the set of objects manipulated by other actions from other processes are *disjoint*. In the case of R_1 and R_2 the atomic accesses to the shared variable x concern single bits.

Executing R_1 and R_2 concurrently may now lead to $x = 3$ – which is the effect of executing the following interleaving of their atomic actions:

$$\begin{array}{l} \textit{first bit of}\ (x) := 0; \ \textit{first bit of}\ (x) := 1; \\ \textit{second bit of}\ (x) := 0; \ \textit{second bit of}\ (x) := 1 \end{array}$$

– or to $x = 2$ – the effect of executing:

$$\begin{array}{l} \textit{first bit of}\ (x) := 1; \ \textit{first bit of}\ (x) := 0; \\ \textit{second bit of}\ (x) := 0; \ \textit{second bit of}\ (x) := 1. \end{array}$$

Other possible interleavings lead to $x = 0$ or $x = 1$.

Secondly consider an alternative implementation of $x := 1$ and $x := 2$ for which $x := 1 \| x := 2$ leads to a different result:

$$\begin{array}{lll} R_1' : & x := 1 \Longleftrightarrow & \textit{first byte of}\ (x) := 00000001; \\ & & \textit{second byte of}\ (x) := 00000000 \\ R_2' : & x := 2 \Longleftrightarrow & \textit{first byte of}\ (x) := 00000010; \\ & & \textit{second byte of}\ (x) := 00000000, \end{array}$$

assuming a word-size for x of two bytes, and *first byte of* $(x) := i$ and *second byte of* $(x) := j$ to be atomic. That is, the atomic accesses to shared variable x concern in the case of R_1' and R_2' single bytes.

Now the effect of executing $R_1' \| R_2'$ is $x = 1$ or $x = 2$, which is different from that of $R_1 \| R_2$ which possibly leads to $x = 3$.

These examples indicate that *to characterise the meaning of a concurrent program one has to specify its atomic actions.*

In the remainder of this section (7.1) we do this by reformulating **Requirement 1** as follows by making a specific choice of atomic actions:

Requirement 1: All accesses to the same shared variable are linearly ordered. There are two ways to access a (shared) variable: by reading its value and by writing its value. In case of reading a value this is considered to be an atomic action, i.e., to take place without being influenced by other processes. When writing a value, this is also considered to be atomic.

Consequently, the atomic accesses of shared variables in the remainder of this section concern the full size of words used for implementing them.

Example 7.1 Let $Q_1 \equiv x := x + 1$, $Q_2 \equiv x := x + 1$ and $Q \equiv Q_1 \parallel Q_2$. Then $\models \{x = 0\}\ Q\ \{x = 2\}$ need not hold, because execution of Q is not equivalent to the sequential execution of $x := x + 1; x := x + 1$, since $x := x + 1$ is not atomic according to **Requirement 1**, reformulated as above. This is explained below. Assume execution of $x := x + 1$ by Q_i to be equivalent to execution of $ti := x; ti := ti + 1; x := ti$ with ti standing for a local register of Q_i (here modelled by a local variable), $i = 1, 2$, and $ti := x$, $ti := ti + 1$ and $x := ti$ considered as atomic actions. Note that this interpretation of the execution of $x := x + 1$ is consistent with **Requirement 1**.

Now a counterexample to the partial correctness formula above is provided by the following interleaving of $ti := x; ti := ti + 1; x := ti$, for $i = 1, 2 : t1 := x; t2 := x; t2 := t2 + 1; x := t2; t1 := t1 + 1; x := t1$, which results in $x = 1$ as postcondition, when started in $x = 0$.

Listing the remaining possible interleavings of $t1 := x; t1 := t1 + 1; x := t1$ and $t2 := x; t2 := t2 + 1; x := t2$ then leads to establishing $\models \{x = 0\}\ Q\ \{x = 1 \vee x = 2\}$. □

7.2 The Generalisation of Floyd's Approach to Nondeterministic Interleavings

The essence of Floyd's inductive assertion method is the observation that a partial-correctness proof can be reduced to checking *finitely* many verification conditions, i.e., for every transition $l \xrightarrow{a} l'$: if execution arrives at l, the associated predicate $\mathcal{Q}_l$ holds, and if the action a is executed, then $\mathcal{Q}_{l'}$ holds at l'. This observation is based upon the fact that every execution sequence is equivalent with a sequence of transitions taken from a fixed, finite, collection.

Since concurrent execution is now modelled by nondeterministic interleavings of atomic actions, the incorporation of concurrency within Floyd's method dictates that every nondeterministic interleaving of atomic actions should be described using such transitions. That is, *execution of a transition should correspond to an atomic action.*

Thus, execution of $P_1 \parallel \ldots \parallel P_n$ should be characterised by the interleaved execution of such transitions from a fixed, finite, collection. Furthermore, during execution of a local transition contained in P_i no execution is modelled in P_j, for $j \neq i$. This implies that a global transition of $P_1 \parallel \ldots \parallel P_n$ contained in P_i is characterised by two n-tuples of *composite* locations, beginning in an n-tuple $\langle l_1, \ldots, l_i, \ldots, l_n \rangle$ and ending in an n-tuple $\langle l_1, \ldots, l_i', \ldots, l_n \rangle$ with $l_1, \ldots, l_n$ and l_i' locations in $P_1, \ldots, P_n$ and P_i, and local transition $l_i \xrightarrow{a} l_i'$ in P_i containing at most one critical reference. In case action a is obvious from the context, $l \xrightarrow{a} l'$ is abbreviated to $l \rightarrow l'$. Composite locations will also be called *global* locations in the following.

Example 7.2 Let x be shared and y be local. Consider the concurrent program fragment illustrated in Figure 1. According to the above, in order to describe the execution of this fragment one has to introduce:

1. composite locations: $\langle s_1, s_2 \rangle$, $\langle s_1, l_2 \rangle$, $\langle s_1, t_2 \rangle$, $\langle t_1, s_2 \rangle$, $\langle t_1, l_2 \rangle$, $\langle t_1, t_2 \rangle$
2. global transitions:

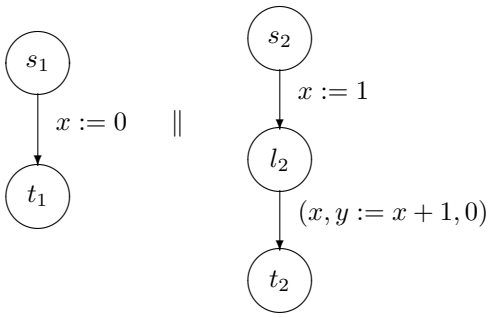

Figure 1: A simple fragment of a shared variable concurrent program.

$$\underbrace{(\langle s_1, s_2\rangle\ ,\ \langle t_1, s_2\rangle)}_{s_2 \text{ fixed}}\ ,\ \underbrace{(\langle s_1, l_2\rangle\ ,\ \langle t_1, l_2\rangle)}_{l_2 \text{ fixed}}\ ,\ \underbrace{(\langle s_1, t_2\rangle\ ,\ \langle t_1, t_2\rangle)}_{t_2 \text{ fixed}}\ ,\ \ldots$$

This leads to the transition diagram in Figure 2. Instead of 3 instructions be-

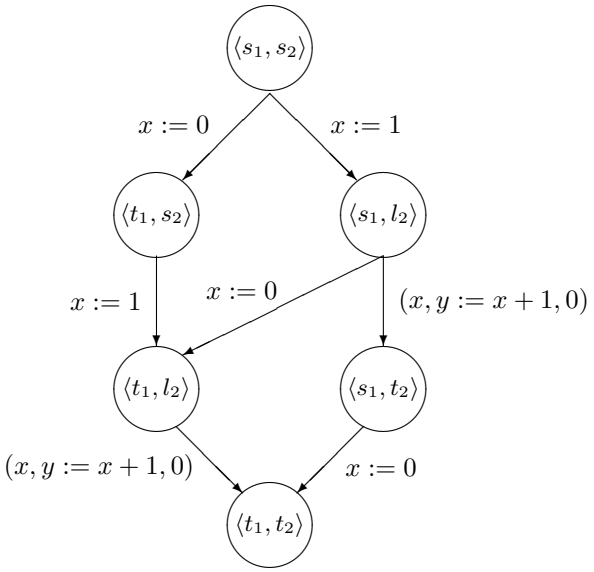

Figure 2: Parallel composition reduced to a sequential diagram.

tween 5 nodes, one has to consider 7 instructions between 6 nodes, which in general leads to an exponential increase of instructions and nodes to be considered. □

Note that we have now considered $(x, y := x + 1, 0)$ as an atomic action.

We continue in the next section with the development of a formal verification theory for concurrent programs operating upon shared variables, based on the assumption that transitions $(l, c \to f, l')$ are executed as single atomic actions.

7.3 Concurrent Transition Systems with Shared Variables

We generalise the inductive assertion method to concurrent transition systems that communicate by means of shared variables. From now on, primitive boolean and state functions are considered to be total, i.e., we base this section on Sessions 1 - 5, unless stated otherwise.

First we define the parallel composition of transition diagrams by means of a transition diagram itself.

Definition 7.3 (Parallel composition) Given transition diagrams $P_i \equiv (L_i, T_i, s_i, t_i)$, $i = 1, \ldots, n$, we define their parallel composition as the product transition diagram $P \equiv (L, T, s, t)$, also denoted by $P_1 \| \ldots \| P_n$, where:

- $L = L_1 \times \ldots \times L_n$
- $T = \{(l, a, l') \mid l = \langle l_1, \ldots, l_i, \ldots, l_n \rangle, l' = \langle l'_1, \ldots, l'_i, \ldots, l'_n \rangle,$ such that $(l_i, a, l'_i) \in T_i, l_j = l'_j, j \neq i\}$
- $s = \langle s_1, \ldots, s_n \rangle$
- $t = \langle t_1, \ldots, t_n \rangle$. □

We can generalise Definition 7.3 to the parallel composition of transition systems by observing that the above definition does not impose any restriction on the names of the nodes involved. Mathematically this amounts to the observation that renaming is a congruence w.r.t. the operation of parallel composition defined above, and that therefore this operation can be extended to the equivalence classes generated by the renaming relation.

In the following we will define all our operations on diagrams. However, in case it is obvious that the renaming relation is a congruence w.r.t. these operations, we will no longer mention their extension to the equivalence classes generated by this relation.

Lemma 7.4 (Associativity and commutativity of parallel composition) Parallel composition of transition systems is associative and commutative. That is, for transition systems P_1, P_2, and P_3, $[[P_1 \parallel P_2] \parallel P_3]$ is one-to-one (i.e., isomorphic) to $[P_1 \parallel [P_2 \parallel P_3]]$, and $[P_1 \parallel P_2]$ is one-to-one (i.e., isomorphic) to $[P_2 \parallel P_1]$.

Proof
Left as an exercise.

■

As a consequence of Lemma 7.4, one can drop brackets inside $P_1 \| \ldots \| P_n$-terms.

Applying the global method

Thus, all definitions based on the notion of transition diagrams in the previous chapter also apply to the parallel composition of transition diagrams (or systems) in the present chapter. In particular, the execution of $P_1 \| \ldots \| P_n$ starts in $s = \langle s_1, \ldots, s_n \rangle$, that is, in all entry nodes of its constituent processes $P_1, \ldots, P_n$. Then execution proceeds by subsequently taking an enabled transition in *one*

of the processes $P_1, \ldots, P_n$. This can take place infinitely often, or until all processes are blocked because there are no enabled transitions left, or until a number of processes have terminated and the remaining ones are blocked, or until the program terminates. In the last case all processes have reached their exit nodes and hence $P_1 \parallel \ldots \parallel P_n$ has reached $t = \langle t_1, \ldots, t_n \rangle$. Otherwise, if a process P_i is no longer enabled and yet t has not been reached, $P_1 \parallel \ldots \parallel P_n$ is said to be *blocked,* and a *deadlock* has been reached.

Note that parallelism is defined by an interleaving of the individual transition of the processes $P_1, \ldots, P_n$. In particular we draw attention to the fact that now the local transitions in the processes P_i,, $i = 1, \ldots, n$, are executed *atomically.*

As in Session 1, the function *val* is defined as follows, where η denotes a maximal execution sequence of $P_1 \parallel \ldots \parallel P_n$:

$$val(\eta) \stackrel{\text{def}}{=} \begin{cases} \sigma, \text{ in case } \eta \text{ terminates in } t, \text{ and the last state of } \eta \text{ is } \sigma, \\ \mathit{fail}, \text{ in case } \eta \text{ ends in } l,\ l \neq t, \text{ to indicate a deadlocked} \\ \quad \text{(or blocked) computation, and} \\ \bot, \text{ in case } \eta \text{ is infinite.} \end{cases}$$

Define *Comp* $[\![P_1 \| \ldots \| P_n]\!]\,\sigma$ as the set of computations of $P_1 \| \ldots \| P_n$ starting in initial state σ, and the *meaning* of $P_1 \| \ldots \| P_n$ as the function:

$$\mathcal{M}\,[\![P_1 \| \ldots \| P_n]\!]\,\sigma \stackrel{def}{=} \{val(\eta) \mid \eta \in Comp\,[\![P_1 \| \ldots \| P_n]\!]\,\sigma\}.$$

Hence $\mathcal{M}\,[\![P_1 \| \ldots \| P_n]\!]\,\sigma$ may contain proper states, and the symbols *fail* and $\bot$ corresponding with nontermination, i.e., with a deadlock and divergence, respectively, which are not considered as proper program states.

The definition of partial correctness from the previous chapter applies immediately to the parallel composition of transition diagrams (or systems) because it results by Definition 7.3 in a particular instance of a transition diagram (or system) according to Definition 1.5.

In particular, a program $P_1 \parallel \ldots \parallel P_n$ is *partially correct* w.r.t. a specification $< \varphi, \psi >$ iff for all states σ, σ', if $\models \varphi(\sigma)$ and $\sigma' \in \mathcal{M}[\![\, P_1 \parallel \ldots \parallel P_n \,]\!](\sigma)$ then $\models \psi(\sigma')$ holds. This is also expressed by $\models \{\varphi\} P_1 \parallel \ldots \parallel P_n \{\psi\}$.

The definitions of success, convergence and total correctness carry over, similarly.

Since the parallel composition of (sequential) transition diagrams is itself a sequential transition diagram, which describes all possible interleavings of the parallel components, we can simply apply the inductive assertion method to concurrent systems. This leads to a *global* proof method where an assertion has to be found for every global location in $P_1 \parallel \ldots \parallel P_n$. If every process P_i has r locations and s edges then $P_1 \parallel \ldots \parallel P_n$ has r^n global locations and $n \times s$ edges. For every edge there are r^{n-1} global locations from which it can start, thus also r^{n-1} verification conditions. Thus for $P_1 \parallel \ldots \parallel P_n$ we have to prove $n \times s \times r^{n-1}$ verification conditions, which is exponential in the number of programs. Consequently this method has no practical value.