Fairness in Parallel Programs: The Transformational Approach

ERNST-RÜDIGER OLDEROG
Christian-Albrechts-Universität Kiel, Kiel
and
KRZYSZTOF R. APT
Centrum voor Wiskunde en Informatica, Amsterdam
and University of Texas at Austin

Program transformations are proposed as a means of providing fair parallelism semantics for parallel programs with shared variables. The transformations are developed in two steps. First, abstract schedulers that implement the various fairness policies are introduced. These schedulers use random assignments $z := ?$ to represent the unbounded nondeterminism induced by fairness. Concrete schedulers are derived by suitably refining the ?. The transformations are then obtained by embedding the abstract schedulers into the parallel programs. This embedding is proved correct on the basis of a simple transition semantics. Since the parallel structure of the original program is preserved, the transformations also provide a basis for syntax-directed proofs of total correctness under the fairness assumption. These proofs make use of infinite ordinals.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Verification

Additional Key Words and Phrases: Fairness, implementation, infinite ordinals, Owicki–Gries method, parallel programs, proof rules, schedulers, shared variables, transformational semantics

1. INTRODUCTION

The study of parallelism is closely connected with the notion of fairness. Let us illustrate this with a simple example. Suppose we are given a function f mapping integers to integers and we wish to search for some zero w of f. A program S_{zero}

A preliminary version of this paper appeared in *Proceedings of the 1st Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 166*, Springer, Berlin, 1984. The research of E.-R. Olderog was partially supported by the German Research Council (DFG) under grant La 426/3-1.

Authors' present addresses: Ernst-Rüdiger Olderog, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Olshausenstr. 40, 2300 Kiel 1, Federal Republic of Germany; Krzysztof R. Apt, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, and Department of Computer Science, University of Texas at Austin, Austin, Texas 78712-1188.

for this task should satisfy the following specification:

$$\{\exists u: f(u) = 0\} S_{\text{zero}} \{f(w) = 0\} \qquad (*)$$

that is, provided f possesses a zero, the variable w will contain such zero upon termination.

A natural solution for S_{zero} is to run two programs in parallel, say S_x and S_y. S_x searches for the zero of f by continuously decrementing a test value x, and S_y by continuously incrementing a test value y. This idea is made precise by the following program:

$S_{\text{zero}} \equiv x := 0;\quad y := 0;$
[**while** $f(x) \neq 0 \wedge f(y) \neq 0$ **do** $x := x - 1$ **od** $\}\ S_x$
‖ **while** $f(x) \neq 0 \wedge f(y) \neq 0$ **do** $y := y + 1$ **od** $\}\ S_y$
];
if $f(x) = 0$ **then** $w := x$ **else** $w := y$ **fi**

But does this program really satisfy the specification (*)? The answer depends critically on the actual meaning of parallel composition. It is easy to see that the usual interpretation in which parallel composition allows any execution order of its components is not sufficient here. For example, S_{zero} might exclusively activate the component S_x, while only S_y could find a zero. What is needed here is a stronger interpretation of parallel composition in which both components S_x and S_y progress.

In this paper we are concerned with this strong interpretation of parallelism. Following [22] and [25], we model it by adding the assumption of fairness. In general, fairness requires that every component of a parallel program S that is "sufficiently often enabled" will eventually progress. For example, if we interpret "sufficiently often enabled" as "not yet terminated," the resulting fairness assumption guarantees that the program S_{zero} will find a zero of f and thus satisfy its specification (*).

But are such fairness assumptions realistic [9]? To discuss this question a low-level view of parallelism using a multiprocessor implementation with full information about the execution times of atomic statements is required. On such a level it is possible to show the proper termination of S_{zero}. Thus the essence of fairness is to provide an appropriate abstraction mechanism from the particular timing conditions of such a multiprocessor implementation.

Unfortunately, this abstraction is not without a price. It implicitly introduces unbounded nondeterminism in the sense that a program will always terminate but with infinitely many possible final states [7, 26]. For example, if one component of the program S_{zero}, say S_y, finds a zero of f, there are, upon termination, infinitely many values possible for the variable x of the other unsuccessful component S_x. It is well known that reasoning about unbounded nondeterminism leads to various complications—various semantic functions lose their continuity [11], and the standard technique of proving loop termination with integer-valued bound functions does not work any more (see, e.g., [3, 21]).

Despite these difficulties, reasoning about unbounded nondeterminism remains a manageable enterprise [1, 3, 7].

So far, semantics and proof theory for fairness assumptions have been studied mainly in the context of nondeterministic **do-od**-programs (see [15] for an overview). For parallel programs S, the question of fairness has been dealt with either by translating S back into a nondeterministic **do-od**-program [5, 21] or by resorting to methods of temporal logic [25] that often require a translation of S into an equivalent formula in temporal logic [22, 29]. But such translations destroy the parallel structure of the original program S and hence the explicit information about its possible concurrency.

In this paper we present a new approach to fairness in parallel programs with shared variables: We provide fair parallelism semantics through transformations that preserve the parallel program structure. These transformations are developed in two steps. First, abstract schedulers are introduced that implement the various fairness policies. "Abstract" means that these schedulers deal only with the fairness of so-called runs, that is, sequences of selections from sets of enabled components. The syntactic form of the components is irrelevant here. "Abstract" also refers to the use of random assignments to represent the unbounded nondeterminism introduced by fairness. A random assignment is of the form $z := ?$ and sets the variable z to some arbitrary nonnegative integer. Concrete schedulers are derived from the abstract ones by choosing some implementation of $z := ?$. We show that our abstract schedulers are both correct and complete; that is, every run checked by the scheduler is fair (correctness), and vice versa, every fair run can be modelled by the scheduler (completeness). Moreover, the schedulers will never cause any deadlock.

The transformations are then obtained by embedding the abstract schedulers into parallel programs. In other words, the application of a transformation T to a program S results in a new program $T(S)$, which can be viewed as S augmented with a built-in scheduler. Only in this second step does our particular choice of program syntax become relevant because T should preserve the parallel structure of S and enjoy further syntactic properties. The correctness of the embedding is proved using a simple transition semantics that models parallelism by interleaving. Thus our analysis of fairness applies only to this model. Further work is needed to extend these results to a model of parallelism in which a number of components can advance simultaneously.

Our approach to fairness in parallel programs derives from the transformation technique for nondeterministic **do-od**-programs in [1] and [2]. However, the separation of abstract schedulers from a concrete program syntax and the development of transformations that preserve the parallel structure of the original program are new features. They allow us to use the transformation T as a syntax-directed method for proving correctness of parallel programs S under fairness assumptions. Since S is correct under the assumption of fairness if and only if the transformed version $T(S)$ is correct in the usual sense, it suffices to prove the correctness of $T(S)$ with (almost) standard methods. In fact, we employ an extension of the Owicki and Gries proof system [24] that makes use of infinite ordinals to deal with termination in the presence of random assignments. As indicated above, such "infinitistic" methods seem necessary in any treatment of unbounded nondeterminism.

Our paper is organized as follows. Section 2 introduces the class of parallel programs studied here, together with the underlying transition semantics. Section 3 defines fair parallelism semantics for three variants of fairness. In Section 4 the two-step transformational approach is developed for the simplest variant of fairness. In Section 5 this approach is extended to the more demanding variants of weak and strong fairness. In Section 6 we present applications of our transformational approach to program correctness. Section 7 concludes our paper by briefly discussing alternative approaches and further developments.

2. PARALLEL PROGRAMS

In this paper we consider parallel programs with shared variables. Their components are sequential programs, that is, usual **while**-programs augmented in our case by random assignments and **await**-statements.

Random assignments have the form $z := ?$ and assign an arbitrary nonnegative integer to z [3]. **await**-statements $S \equiv$ **await** B **then** S_1 **end** are used to achieve synchronization in the context of parallel composition. S is executed only if B is true. In this case S is executed as an indivisible action [24]. **await**-statements cannot be nested.

Formally, a parallel program has the form

$$S \equiv S_0; [S_1 \parallel \cdots \parallel S_n]$$

where S_0 is a sequence of assignments and $S_1, \ldots, S_n$ are sequential programs. S_0 is the *initial part* of S, and $S_1, \ldots, S_n$ are the *components* of S inside the *parallel composition* $[S_1 \parallel \cdots \parallel S_n]$. We distinguish four classes of parallel programs: $L(\parallel)$, $L(\parallel, ?)$, $L(\parallel,$ **await**$)$, and $L(\parallel,$ **await**, $?)$ depending on whether random assignments or/and **await**-statements are used. $L(\parallel,$ **await**$)$ is essentially the language studied in [24].

In this paper we shall study certain program transformations, that is, mappings

$$T: L(\parallel)[\text{or } L(\parallel, \textbf{await})] \rightarrow L(\parallel, \textbf{await}, ?).$$

In order to prove the correctness of such transformations we need a rigorous semantics of parallel programs. We choose here a particularly simple semantics following the style of [17].

We assume that all variables are of type *integer* or *Boolean*. Thus programs are executed over a domain consisting of all integers and {**true**, **false**} with the usual operations available. A (*proper*) *state* is a function assigning to each variable a value of the appropriate type from the domain.

We use the following notation: A typical domain element is denoted by the letter d; Var is the set of variables with typical elements x, y, z; Var(S) denotes the set of variables occurring in a program S; Σ is the set of proper states with typical elements σ, τ. As usual, $\sigma[d/x]$ is a state variant that agrees with σ, except for the variable x where the value is d, $\sigma(B)$ and $\sigma(t)$ are the values of a Boolean expression B or a term t in the state σ, and $\sigma \restriction X$ is the restriction of σ to the set X of variables. We also need two special states not present in Σ: $\perp$ reporting divergence and Δ reporting deadlock.

By a *configuration* we mean a pair $\langle S, \sigma \rangle$ consisting of a program $S \in L(\parallel,$ **await**, $?)$ and a state σ. Following [17] and [28] we introduce a *transition relation* $\rightarrow$ between these configurations. $\langle S, \sigma \rangle \rightarrow \langle S_1, \sigma_1 \rangle$ means: Executing S

one step in σ can lead to σ_1, with S_1 being the remainder of S still to be executed. To express termination we allow (in configurations only) the empty program E with E; $S \equiv S$; $E \equiv S$. As usual $\rightarrow^*$ denotes the reflexive, transitive closure of $\rightarrow$.

The relation $\rightarrow$ is defined by structural induction on $L(\|, \textbf{await}, ?)$:

$\langle \text{skip}, \sigma \rangle \rightarrow \langle E, \sigma \rangle$,
$\langle x := t, \sigma \rangle \rightarrow \langle E, \sigma[d/x] \rangle$ if $\sigma(t) = d$,
$\langle z := ?, \sigma \rangle \rightarrow \langle E, \sigma[d/z] \rangle$ for every $d \geq 0$,
$\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle$ if $\sigma(B) = \textbf{true}$,
$\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi } \sigma \rangle \rightarrow \langle S_2, \sigma \rangle$ if $\sigma(B) = \textbf{false}$,
$\langle \textbf{while } B \textbf{ do } S_1 \textbf{ od}, \sigma \rangle \rightarrow \langle S_1; \textbf{while } B \textbf{ do } S_1 \textbf{ od}, \sigma \rangle$ if $\sigma(B) = \textbf{true}$
$\langle \textbf{while } B \textbf{ do } S_1 \textbf{ od}, \sigma \rangle \rightarrow \langle E, \sigma \rangle$ if $\sigma(B) = \textbf{false}$,
if $\langle S_1, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$ and $\sigma(B) = \textbf{true}$ then $\langle \textbf{await } B \textbf{ then } S_1 \textbf{ end}, \sigma \rangle \rightarrow \langle E, \tau \rangle$,
if $\langle S_1, \sigma \rangle \rightarrow \langle S_2, \tau \rangle$ then $\langle S_1; S, \sigma \rangle \rightarrow \langle S_2; S, \tau \rangle$
if $\langle S_i, \sigma \rangle \rightarrow \langle T_i, \tau \rangle$ for some $i \in \{1, \ldots, n\}$ then
$\langle [S_1 \| \cdots \| S_i \| \cdots \| S_n], \sigma \rangle \rightarrow \langle [S_1 \| \cdots \| T_i \| \cdots \| S_n], \tau \rangle$.

As demonstrated above, skip statements, assignments, evaluations of Boolean expressions, and **await**-statements are executed in one step, that is, as atomic or indivisible actions. Therefore statements of the form skip, $x := t$, $z := ?$, and **await** B **then** S_1 **end** are called *atomic*. Parallel composition is modelled by interleaving the transitions of its components.

Based on $\rightarrow$ we introduce some further concepts. A configuration $\langle S, \sigma \rangle$ is *maximal* if it has no successor with respect to $\rightarrow$. A *terminal* configuration is a maximal configuration $\langle S, \sigma \rangle$ with $S \equiv [E \| \cdots \| E]$. All other maximal configurations are called *deadlocked*. A *computation* of S (*starting* in σ) is a finite or infinite sequence of the form

$$\xi\colon \langle S, \sigma \rangle \rightarrow \langle S_1, \sigma_1 \rangle \rightarrow \cdots \rightarrow \langle S_k, \sigma_k \rangle \cdots$$

A computation of S is called *terminating* (*deadlocking*) if it is of the form

$$\xi\colon \langle S, \sigma \rangle \rightarrow \cdots \rightarrow \langle T, \tau \rangle$$

where $\langle T, \tau \rangle$ is terminal (deadlocked). Infinite computations of S are called *diverging*. We say that S *can diverge from* σ (*can deadlock from* σ) if there exists a diverging (deadlocking) computation of S starting in σ.

The *parallelism semantics* of programs $S \in L(\|, \textbf{await}, ?)$ is a mapping

$$\mathcal{M}[\![S]\!]\colon \Sigma \rightarrow \mathcal{P}(\Sigma \cup \{\bot, \Delta\})$$

defined by

$$\mathcal{M}[\![S]\!](\sigma) = \{\tau \mid \langle S, \sigma \rangle \rightarrow^* \langle [E \| \cdots \| E], \tau \rangle\} \quad (1)$$
$$\cup \{\bot \mid S \text{ can diverge from } \sigma\} \quad (2)$$
$$\cup \{\Delta \mid S \text{ can deadlock from } \sigma\} \quad (3)$$

where $\mathcal{P}(X)$ denotes the powerset of a given set X. Thus $\mathcal{M}[\![S]\!]$ assigns to every initial state σ the set of possible final states (including $\bot$ and Δ) resulting from computations of S.

If a terminating program has only finitely many possible finals states, it exhibits *bounded nondeterminism*; otherwise it exhibits *unbounded nondetermin-*

ism [7]. For example, the random assignment $z := ?$ exhibits unbounded nondeterminism. Under the semantics $\mathcal{M}$ all programs without random assignments exhibit only bounded nondeterminism. Formally, we have

LEMMA 2.1 (Bounded Nondeterminism). *Let S be an $L(\|, \textbf{await})$-program and σ be a state. Then $\mathcal{M}[\![S]\!](\sigma)$ is finite or it contains $\perp$.*

PROOF The set of computation sequences of S starting in σ can be represented as a finitely branching computation tree. By König's lemma, this tree is finite or it contains an infinite path. Lemma 2.1 now follows immediately. □

Of course, changing the semantics $\mathcal{M}$ may invalidate the lemma. Such a change will be discussed in the next section. Note that $\mathcal{M}$ identifies all infinite computations with divergence. This identification is justified since we are interested in terminating programs. However, in Section 7 we briefly discuss a process semantics Π which considers infinite computations.

Some further notions will be helpful. The ith parallel component has *terminated* in $\langle [S_1 \| \cdots \| S_n], \sigma \rangle$ if $S_i \equiv E$; it is *disabled* in $\langle [S_1 \| \cdots \| S_n], \sigma \rangle$ if either $S_i \equiv E$ or $S_i \equiv \textbf{await } B \textbf{ then } S \textbf{ end}$; T with $\sigma(B) = \textbf{false}$; it is *enabled* if it is not disabled, that is, if S_i is not terminated and whenever $S_i \equiv \textbf{await } B \textbf{ then } S \textbf{ end}$; T holds, then $\sigma(B) = \textbf{true}$. The ith component is *active* in the step $\langle [S_1 \| \cdots \| S_n], \sigma \rangle \rightarrow \langle [T_1 \| \cdots \| T_n], \tau \rangle$ if $S_i \neq T_i$. A program S is *deadlock-free* if $\Delta \notin \mathcal{M}[\![S]\!](\sigma)$ for every state σ.

3. FAIRNESS

The parallelism semantics $\mathcal{M}$ interprets parallel composition as the indication that any execution order of its components is acceptable. This leaves maximal freedom to the implementor of parallel composition, but often this is not what we wish to express when writing the symbol $\|$.

Consider, for example, the program

$$S^* \equiv \underbrace{[\textbf{while } b \textbf{ do } x := x + 1 \textbf{ od}}_{S_1} \| \underbrace{b := \textbf{false}]}_{S_2}$$

Under the parallelism semantics $\mathcal{M}$, it can diverge since it may exclusively activate its first component S_1. But under a multiprocessor implementation of parallelism the second component S_2 is eventually executed; this causes termination of S^*.

To abstract from the details of multiprocessor implementations, the notion of fairness leading to a fair parallelism semantics is used. Since fairness can be defined exclusively in terms of enabled and activated components, we abstract from all other details in computations and introduce the notions of selection and run. This will simplify our subsequent analysis of fairness.

A *selection* (*of n components*) is a pair (E, i) consisting of a nonempty set $E \subseteq \{1, \ldots, n\}$ of enabled components and an activated component $i \in E$. A *run* (*of n components*) is a finite or infinite sequence

$$(E_0, i_0)(E_1, i_1) \cdots (E_j, i_j) \cdots$$

of selections. A run is called *monotonic* if

$$E_0 \supseteq E_1 \supseteq \cdots \supseteq E_j \supseteq \cdots$$

holds.

We can now define fairness. A run is called *weakly fair* if it satisfies the following condition:

$$\forall i \in \{1, \ldots, n\}\colon ((\overset{\infty}{\forall} j \in \mathbb{N}_0\colon i \in E_j) \rightarrow (\overset{\infty}{\exists} j \in \mathbb{N}_0\colon i = i_j)).$$

The quantifier $\overset{\infty}{\forall}$ means "for all, but finitely many" or "from a certain moment on" and $\overset{\infty}{\exists}$ stands for "there exist infinitely many." $\mathbb{N}_0$ denotes the set {0, 1, 2, 3, ...}. Thus in a weakly fair run every component i, which is almost always enabled, is activated infinitely often.

A run is called *strongly fair* if it satisfies the following condition:

$$\forall i \in \{1, \ldots, n\}\colon ((\overset{\infty}{\exists} j \in \mathbb{N}_0\colon i \in E_j) \rightarrow (\overset{\infty}{\exists} j \in \mathbb{N}_0\colon i = i_j)).$$

Thus in a strongly fair run, every component i that is infinitely often enabled is activated infinitely often. Since for monotonic runs both notions of fairness coincide, we simply talk of *fair* runs in this case. Note that, by definition, finite runs are always weakly and strongly fair.

The distinction between fairness and weak and strong fairness is taken from [1]: It corresponds to the distinction among impartiality, justice, and fairness in [21], though in general fairness and impartiality differ.

Next we link runs to computations. Consider a parallel program $S \equiv S_0; [S1 \parallel \cdots \parallel S_n]$. The *run of a computation*

$$\begin{aligned} \langle S, \sigma \rangle &\equiv \langle S_0; [S_1 \parallel \cdots \parallel S_n], \sigma \rangle \\ &\quad \rightarrow \cdots \rightarrow \langle E; [S_1 \parallel \cdots \parallel S_n], \sigma_0 \rangle \qquad (1) \\ &\equiv \langle [S_1 \parallel \cdots \parallel S_n], \sigma_0 \rangle \\ &\equiv \langle T_0, \sigma_0 \rangle \rightarrow \cdots \rightarrow \langle T_j, \sigma_j \rangle \cdots \qquad (2) \end{aligned}$$

of S is defined as the run

$$(E_0, i_0)(E_1, i_1) \cdots (E_j, i_j) \cdots$$

where for every $j \geq 0$

$$E_j = \{i \mid \text{the } i\text{th component is enabled in } \langle T_j, \sigma_j \rangle\}$$

provided $E_j \neq \varnothing$ and the i_jth component is active in the step $\langle T_j, \sigma_j \rangle \rightarrow \langle T_{j+1}, \sigma_{j+1} \rangle$.

In this definition it is understood that the transition steps in line (1) just serve to completely execute the initial part S_0 of S. Thus, computations that do not reach the end of S_0 yield the empty run. A *run of a program* S is the run of one of its computations.

For $L(\parallel)$-programs we have the following simplification:

LEMMA 3.1 (Monotonicity). *Every run of an $L(\parallel)$-program is monotonic.*

PROOF. A component of an $L(\parallel)$-program is enabled iff it is not terminated. Thus whenever a component i gets disabled in a run ($i \notin E_j$) it remains disabled ($\forall k \geq j: i \notin E_k$). This implies the monotonicity of the runs. □

We define a *computation* to be *weakly* (*strongly*) *fair* if its run is weakly (strongly) fair. By Lemma 3.1, weak and strong fairness coincide for computations of $L(\|)$-programs; hence we talk of *fair* computations in this case. Each notion of fairness leads to a corresponding *fair parallelism semantics*. For weak fairness we obtain

$$\begin{aligned}&\mathcal{M}_{\text{wfair}}[\![S]\!](\sigma)\\ &\quad = \{\tau \mid \langle S, \sigma\rangle \to \cdots \to \langle [E \parallel \cdots \parallel E], \tau\rangle\} && (1)\\ &\qquad \cup \{\bot \mid \exists \text{ infinite } \textit{weakly fair} \text{ computation of } S \text{ starting in } \sigma\} && (2)\\ &\qquad \cup \{\Delta \mid S \text{ can deadlock from } \sigma\} && (3)\end{aligned}$$

Analogous are the definitions of $\mathcal{M}_{\text{sfair}}[\![S]\!](\sigma)$ and, for $S \in L(\|)$ only, of $\mathcal{M}_{\text{fair}}[\![S]\!](\sigma)$. Note that $L(\|)$-programs cannot deadlock: hence $\Delta \notin \mathcal{M}_{\text{fair}}[\![S]\!](\sigma)$.

To exercise these definitions, let us look at the above $L(\|)$-program S^* again. A computation of S^* that exclusively activates the first component S_1 is not fair because in its run

$$(\{1, 2\}, 1)(\{1, 2\}, 1) \cdots (\{1, 2\}, 1) \cdots$$

the number 2, that is, the second component S_2, is never activated. Thus in any fair computation of S^* the second component S_2 of S^* is activated at least once, setting b to **false**. This will cause termination of the **while**-loop of the first component S_1 and hence the program S^* itself. Summarizing, under the fair parallelism semantics, S^* always terminates. Formally,

$$\bot \notin \mathcal{M}_{\text{fair}}[\![S^*]\!](\sigma) \qquad \text{for every state } \sigma.$$

Observe, however, that there are infinitely many final states possible for S^*. For example, starting in σ with $\sigma(x) = 0$ we obtain

$$\mathcal{M}_{\text{fair}}[\![S^*]\!](\sigma) = \{\sigma[0/x], \sigma[1/x], \sigma[2/x], \ldots\}.$$

This is because it is not known how often the assignment $x := x + 1$ of S_1 is executed before S_2 sets b to **false**. Thus, by assuming fair-parallelism, even programs without random assignment can exhibit unbounded nondeterminism, in contrast to the Bounded Nondeterminism Lemma 2.1 for ordinary parallelism.

4. TRANSFORMATIONAL SEMANTICS IN $L(\|)$

For parallel composition we have introduced two types of interpretation: ordinary parallelism and fair parallelism. The latter was obtained by restricting the set of computations. This provides a clear definition of fair parallelism but no insight into dealing with it in terms of implementation or correctness proofs. We wish to provide such an insight by applying the *principle of transformational semantics*: Reduce the new concept to known concepts with the help of program transformations.

In this section we restrict ourselves to programs in $L(\|)$. Hence our aim is to find a transformation T that reduces the fair parallelism semantics $\mathcal{M}_{\text{fair}}$ of $L(\|)$-programs S to the usual parallelism semantics $\mathcal{M}$ in the sense that

$$\mathcal{M}_{\text{fair}}[\![S]\!] = \mathcal{M}[\![T(S)]\!].$$

Note that we cannot expect the transformed program $T(S)$ to be in $L(\|)$ again because $\mathcal{M}_{\text{fair}}$ introduces unbounded nondeterminism as opposed to $\mathcal{M}$ (cf.

Lemma 2.1 and Section 3). But we can localize this unbounded nondeterminism by using random assignments $z := ?$ in $T(S)$. T will be useful as a basis for syntax-directed correctness proofs of parallel programs under the assumption of fairness (see Section 6). T will also shed light on the possible implementation of fair parallelism because it can be seen as embedding into the original program S a scheduler allowing only fair computations.

We begin with a general result about fair schedulers using the abstract notions of selection and run.

4.1 Schedulers

Following [12], a scheduler is an automaton that enforces a certain fairness policy on the computations of a parallel program S. To this end, the scheduler keeps in its local state sufficient information about the run of a computation and engages in the following interaction with the program.

At certain moments during a computation the program presents the set E of currently enabled components to the scheduler (provided $E \neq \emptyset$). By consulting its local state the scheduler returns to the program a nonempty subset I of E, namely, the set of components that, upon activation in the next transition step, will still satisfy the fairness policy. Now the program selects one component $i \in I$ for activation, and the scheduler updates its local state accordingly.

From a more abstract point of view, we may ignore the actual interaction between program and scheduler and just record the result of this interaction, namely, the selection (E, i) checked by the scheduler. Summarizing, we arrive at the following definition:

A *scheduler SCH* (*for n components*) is given by

—a set of local *scheduler states* σ, which are disjoint from the program states;

—a subset of *initial* scheduler states; and

—a *scheduling relation*

$$\text{sch} \subseteq \{\text{scheduler states}\} \times \{\text{selections of } n \text{ components}\} \times \{\text{scheduler states}\}$$

which is *deadlock-free*, that is,

$$\forall \sigma \forall E \neq \emptyset \quad \exists i \in E \quad \exists \sigma' \colon \text{sch}(\sigma, (E, i), \sigma').$$

The term "deadlock-free" for sch is justified because the scheduler will never cause any (additional) deadlock in the program: for every scheduler state σ and every nonempty set E of enabled components there exists a component $i \in E$ such that the selection (E, i) together with the updated local state σ' satisfy the scheduling relation.

Consider now a finite or infinite run

$$(E_0, i_0)(E_1, i_1) \cdots (E_j, i_j) \cdots \qquad (*)$$

and a scheduler SCH. We wish to ensure that sufficiently many, but not necessarily all selections (E_j, i_j) are checked by SCH. To this end, we take a so-called *check-set*

$$\mathscr{C} \subseteq \mathbb{N}_0$$

representing the positions of selections to be checked, and say that the run (*) can be checked by SCH at the positions in $\mathscr{C}$ if there exists a finite or infinite sequence

$$\sigma_0\sigma_1 \cdots \sigma_j \cdots$$

of scheduler states, with σ_0 being an initial scheduler state, such that for all $j \geq 0$

$$\text{sch}(\sigma_j, (E_j, i_j), \sigma_{j+1}) \qquad \text{if} \quad j \in \mathscr{C},$$

and

$$\sigma_j = \sigma_{j+1} \qquad \text{otherwise.}$$

Thus for $j \in \mathscr{C}$ the scheduling relation sch checks the selection (E_j, i_j) made in the run using and updating the current scheduler state; for $j \notin \mathscr{C}$ there is no interaction with the scheduler and hence the current scheduler state remains unchanged (for technical convenience, however, this is treated as an identical step $\sigma_j = \sigma_{j+1}$).

For example, with $\mathscr{C} = \{2n + 1 \mid n \in \mathbb{N}_0\}$ every second selection in (*) is checked. This can be pictured as follows:

Run:		(E_0, i_0)		(E_1, i_1)		(E_2, i_2)		(E_3, i_3)	
				⇵		·		⇵	
Scheduler:	σ_0	=	σ_1	SCH	σ_2	=	σ_3	SCH	···

Using the programming notation of Section 2 we present now a specific scheduler FAIR. For n components it is defined as follows:

—the scheduler state is given by n integer variables $z_1, \ldots, z_n$,

—this state is initialized nondeterministically by the random assignments

$$\text{INIT} \equiv z_1 := ?; \cdots; z_n := ?,$$

—the scheduling relation $\text{sch}(\sigma, (E, i), \sigma')$ holds iff σ, E, i, and σ' are as follows:

(i) σ is given by the current values of $z_1, \ldots, z_n$.

(ii) E and i satisfy the condition

$$\text{SCH}_i \equiv z_i = \min\{z_k \mid k \in E\}.$$

(iii) σ' is obtained from σ by executing

$\text{UPDATE}_i \equiv z_i := ?;$
for all $j \in \{1, \ldots, n\} - \{i\}$ **do**
if $j \in E$ **then** $z_j := z_j - 1$ **fi**
od

where the **for**-statement

for all $j \in \{1, \ldots, n\} - \{i\}$ **do** S_j **od**

abbreviates

$$S_1; \ldots; S_{i-1}; S_{i+1}, \ldots, S_n.$$

How does FAIR work? The scheduling variables $z_1, \ldots, z_n$ represent *priorities* assigned to the n components of a parallel program. A component i has a higher priority than a component j if $z_i < z_j$. Initially, the components get arbitrary priorities. If, during a run, FAIR is presented with a set E of enabled components, it selects a component $i \in E$ that has maximal priority, that is, with

$$z_i = \min\{z_k \mid k \in E\}.$$

Note that for any nonempty set E and any values of $z_1, \ldots, z_n$ there exists some $i \in E$ with this property. Thus the scheduling relation $\mathrm{sch}(\sigma, (E, i), \sigma')$ of FAIR is deadlock-free as required.

The update of the scheduling variables guarantees that the priorities of all enabled but not selected components j get increased (by decrementing z_j by 1). The priority of the selected component i, however, gets reset arbitrarily. The idea is that by gradually increasing the priority of enabled components j they cannot be refused forever. The following theorem makes this idea precise.

THEOREM 4.1 (FAIR for Monotonic Run). *Consider a monotonic run. This run is fair iff it can be checked by FAIR at the positions in an arbitrary infinite check set* $\mathscr{C}$.

In other words, no infinite suffix of the run may be left unchecked by FAIR. This formulation leaves a lot of freedom for choosing the actual check set $\mathscr{C}$, a freedom we shall exploit when embedding the scheduler FAIR into parallel programs. Theorem 4.1 states correctness and completeness of FAIR. *Correctness* means that every monotonic run checked by FAIR is indeed fair; *completeness* means that every fair monotonic run can be checked by FAIR.

PROOF OF THEOREM 4.1. Consider a monotonic run

$$(E_0, i_0) \cdots (E_j, i_j) \cdots \qquad (*)$$

of n components and an infinite check set $\mathscr{C}$.

If: Let $(*)$ be checked at the positions in $\mathscr{C}$, that is, let there be a sequence $\sigma_0 \cdots \sigma_j \cdots$ of states of FAIR satisfying $\mathrm{sch}(\sigma_j, (E_j, i_j), \sigma_{j+1})$ for $j \in \mathscr{C}$ and $\sigma_j = \sigma_{j+1}$ otherwise. We show that $(*)$ is fair.

Suppose the contrary. Then $(*)$ is infinite, and by its monotonicity there exists some component $i \in \{1, \ldots, n\}$ which from some moment $j \geq 0$ on is always enabled but never activated, that is,

$$\forall k \geq j : i \in E_k \wedge i \neq i_k.$$

Since $(*)$ is checked infinitely often, the variable z_i of FAIR, which gets decremented by each check, becomes arbitrarily small, in particular smaller than $-n$ in some state σ_k with $k \geq j$. But this is impossible because the assertion

$$\mathrm{INV} = \bigwedge_{k=1}^{n} \mathrm{card}\{i \mid z_i \leq -k\} \leq n - k$$

holds in every state σ_j of FAIR. Here card M denotes the cardinality of a set M.

We prove this invariant by induction on $j \geq 0$. In σ_0 we have $z_1, \ldots, z_n \geq 0$ so that INV is trivially satisfied. Assume now that INV holds in σ_j. We show that INV is also true in σ_{j+1}. Suppose INV is false in σ_{j+1}. Then there is some $k \in \{1, \ldots, n\}$ such that there are at least $n - k + 1$ indices i for which $z_i \leq -k$ holds in σ_{j+1}. Let I be the set of all these indices. Thus, card $I \geq n - k + 1$. By the definition of FAIR, $z_i \leq -k + 1$ holds for all $i \in I$ in σ_j. Thus, card $I \leq n - k + 1$ by the induction hypothesis. So actually, card $I = n - k + 1$ and

$$I = \{i \mid z_i \leq -k + 1 \text{ holds in } \sigma_j\}.$$

Since INV holds in σ_j but not in σ_{j+1}, we conclude $\text{sch}(\sigma_j, (E_j, i_j), \sigma_{j+1})$, that is, the position j of the run $(*)$ is checked by FAIR. By the definition of FAIR, the activated component i_j is in I. But this is a contradiction because $z_{i_j} \geq 0$ holds in σ_{j+1} by the UPDATE_{i_j} part of FAIR. Thus INV remains true in σ_{j+1}.

Only If: Conversely, let the run $(*)$ be fair. We show that $(*)$ can be checked at the positions in $\mathscr{C}$ by constructing a sequence $\sigma_0 \cdots \sigma_j \cdots$ of states of FAIR satisfying $\text{sch}(\sigma_j, (E_j, i_j), \sigma_{j+1})$ for $j \in \mathscr{C}$ and $\sigma_j = \sigma_{j+1}$ otherwise. The construction proceeds by assigning appropriate values to the variables $z_1, \ldots, z_n$ of FAIR. For $i \in \{1, \ldots, n\}$ and $j \in N_0$ we put

$$\sigma_j(z_i) = 1 + \text{card}\{l \in \mathscr{C} \mid j \leq l < m_{i,j} \wedge i_l \neq i\}$$

where

$$m_{i,j} = \min\{m \in \mathscr{C} \mid j \leq m \wedge (i \notin E_m \vee i_m = i)\}.$$

Note that $\min m_{i,j} \in \mathbb{N}_0$ exists because the run $(*)$ is monotonic and fair; informally, $\sigma_j(z_i)$ is 1 + the number of times the component i is neglected during checked selections ($i_l \neq i$) before its termination ($i \notin E_m$) or its own next checked selection ($i_m = i$). Note that in every state σ_j the variables $z_1, \ldots, z_n$ have values ≥ 1 and exactly one variable z_i has the value 1, the one which is activated next. It is easy to see that this assignment of values $\sigma_j(z_i)$ is possible with FAIR. This completes the desired construction. □

Discussion. As indicated above, our view of a scheduler is close to the definition in [12]. The differences are as follows:

(1) our schedulers need not check every selection (E, i) in a run, and
(2) our schedulers may be nondeterministic in their choice of which component $i \in E$ to activate next.

The first point allows an efficient embedding of FAIR into parallel programs later in Section 4.2; the second point was used when proving the completeness part of Theorem 4.1, that is, every fair run can be checked by FAIR. Consequently, we can obtain every other fair scheduler for monotonic runs by implementing the nondeterministic choices in FAIR. Due to [10] and [26], implementing nondeterminism means *narrowing* the set of nondeterministic choices. Thus a random assignment $z := ?$ can be implemented by any ordinary assignment $z := t$ where t yields values ≥ 0.

Consider, for example, a simple *round robin scheduler* RORO which selects the enabled components clockwise (see, e.g., [31]). RORO enforces fairness in $L(\|)$-programs. Starting from FAIR, it can be implemented by replacing the

random assignments inside INIT and UPDATE$_i$ as follows:

$$\text{INIT} \equiv z_1 := 1; z_2 := 2; \ldots; z_n := n$$

and

$$\text{UPDATE}_i \equiv z_i := n; \quad \textbf{for all} \cdots \textbf{do} \cdots \textbf{od}.$$

In monotonic runs RORO always schedules the next enabled component in the cyclic ordering $1, 2, \ldots, n$.

Clearly, this implementation is too expensive in terms of storage requirements. Since we need to remember only which component i is to be selected next, the variables $z_1, \ldots, z_n$ of RORO can be condensed into one variable z ranging over $\{1, \ldots, n\}$ and pointing to the index of the chosen component. The resulting implementation is given in [27]. It uses only n scheduler states; as shown in [12] this number is optimal for (weakly) fair schedulers for n components.

In an early note Dijkstra [10] investigates deadlock-free strategies to avoid starvation among competing processes. For each process i, a fixed a priori bound N_i is postulated such that process i should never be delayed more than N_i times. For given delays $N_i \geq n - 1$ and $i = 1, \ldots, n$, Dijkstra's strategy can be viewed as a scheduler STRAT obtained from FAIR by implementing the random assignments as follows:

$$\text{INIT} \equiv z_1 := N_1; \ldots; z_n := N_n,$$

$$\text{UPDATE}_i \equiv z_i := N_i; \quad \textbf{for all} \cdots \textbf{do} \cdots \textbf{od}.$$

The scheduling variables $z_1, \ldots, z_n$ are called *allowance counts* in [10]. It is shown that in every run of STRAT these variables form a so-called *safe* set, that is, satisfy the invariant

$$\text{SAFE} \equiv \forall k \geq 0 : \text{card}\{i \mid z_i \leq k\} \leq k.$$

Dijkstra considers only fixed bounded delays N_i and no unbounded delays needed to model fairness.

Finally, let us point out an alternative formulation of the updates in FAIR. Following [27], we could have chosen simply

$$\text{UPDATE}_i \equiv z_i := z_i + 1 + ?$$

where, as before, ? stands for an arbitrary nonnegative integer value and Theorem 4.1 would remain valid. This way of updating z_i resembles the construction used in Lamport's "bakery algorithm" for mutual exclusion [18]. Its advantage is that embedding FAIR into a parallel program would then lead to a so-called *distributed* solution in which each variable z_i can be modified only by one component (but read by any number of components). We did not adopt this solution because it yields unbounded values of the variable z_i in any implementation of ?, contrary to our definition of UPDATE$_i$. A similar problem arises in [18].

4.2 Transformations

We return to the problem of finding a program transformation T which for every $L(\|)$-program S reduces fair parallelism $\mathcal{M}_{\text{fair}}$ to asynchronous parallelism $\mathcal{M}$ in

the sense of transformational semantics:

$$\mathcal{M}_{\text{fair}}[\![S]\!] = \mathcal{M}[\![T(S)]\!].$$

The idea is to obtain T by embedding the scheduler FAIR into S. This task is simplified by the fact that FAIR is given in programming notation, but it is not obvious which form the embedding should take. We discuss the possibilities.

First attempt. A simple way of reducing fairness is via nondeterministic programs. Given a parallel program $S \equiv S_0; [S_1 \| \cdots \| S_n]$ one first follows the approach of [13], [14], [8], or [9] and translates S into a nondeterministic **do-od**-program [11], for example, of the form

$T_{\text{ndet}}(S) \equiv S_0$; **do** enabled$_1 \rightarrow$ execute S_1 one step
:
□ enabled$_n \rightarrow$ execute S_n one step
od.

Next, one can apply the transformations T_{fair} of [1] or [5], which use random assignments to realize the assumption of fairness in the context of **do-od**-programs. Roughly, T_{fair} of [1] can be understood as an embedding of the scheduler FAIR into **do-od**-programs. This embedding is very simple: If

$S' \equiv$ **do** $B_1 \rightarrow S_1' \,\square \cdots \square\, B_n \rightarrow S_n'$ **od**

then

$T_{\text{fair}}(S') \equiv$ INIT; **do** $B_1 \wedge \text{SCH}_1 \rightarrow \text{UPDATE}_1; S_1'$
:
□ $B_n \wedge \text{SCH}_n \rightarrow \text{UPDATE}_n; S_n'$
od

where INIT, SCH$_i$, and UPDATE$_i$ come from FAIR.

Combining both transformations yields

$$T(S) = T_{\text{fair}}(T_{\text{ndet}}(S)).$$

Studying parallelism through nondeterminism is a valid and often pursued approach. The drawback is that the translation T_{ndet} destroys the parallel structure of the original program S and hence the explicit information about its possible concurrency.

We therefore aim at a transformation that preserves the structure of S. For example, we could add the scheduler FAIR as an extra component to S, roughly yielding

$$T(S) \equiv S_0; [\text{FAIR} \| S_1 \| \cdots \| S_n].$$

However, this transformation does not support syntax-directed correctness proofs of S under the assumption of fairness. Instead we want a transformation that distributes the scheduler FAIR over the components $S_1, \ldots, S_n$ of S and therefore state the following definition:

Definition 4.2. A transformation $T: L(\|) \rightarrow L(\|, \textbf{await}, ?)$ is called $\|$-*preserving* if T satisfies

$$T(S_0; [S_1 \| \cdots \| S_n]) = T_0^n(S_0); [T_1^n(S_1) \| \cdots \| T_n^n(S_n)]$$

where T_i^n is a subtransformation working on the ith component of S. The notation suggests that the only information T_i^n may use about the structure of S is the total number n of components in S and the index i of the currently transformed component.

Second attempt. As illustrated above, embedding FAIR into nondeterministic **do-od**-programs changes the program structure in a rather modest way. Essentially, only random assignments are added and Boolean expressions are refined. Correspondingly, we might expect a similar type of transformation $T: L(\|) \to L(\|, ?)$.

Definition 4.3. A transformation $T: L(\|) \to L(\|, ?)$ is *schematic* if it is $\|$-preserving and if for every $S \in L(\|)$ there is a set Z of new auxiliary variables $z \in Z$ used in $T(S)$ for scheduling purposes in the following two ways:

(1) in additional assignments of the form $z := ?$ or $z := t$ (possibly conditional) inside of S, and

(2) in Boolean conjuncts c used to strengthen Boolean expressions b of loops or conditionals in S. We require that this strengthening is done schematically, that is, the conjunct c is independent of the actual form of b.

Note that, since $T(S)$ manipulates additional variables Z, the best we can hope to prove is that $\mathcal{M}_{\text{fair}}[\![S]\!]$ agrees with $\mathcal{M}[\![T(S)]\!]$ "modulo Z," that is, that the states they produce agree on all variables except those in Z. Surprisingly, the following theorem holds.

THEOREM 4.4. *There is no schematic transformation* $T: L(\|) \to L(\|, ?)$ *such that for every program* $S \in L(\|)$

$$\mathcal{M}_{fair}[\![S]\!] = \mathcal{M}[\![T(S)]\!] mod\ Z$$

holds where Z *is the set of auxiliary variables used in* $T(S)$.

PROOF. Consider the following program

$S \equiv$ [**while** b **do** skip **od**; $b :=$ **true** $\|$ $b :=$ **false**].

According to the $\mathcal{M}_{\text{fair}}$ semantics, S always terminates. Suppose, by contradiction, that a transformation T satisfying the claim of the theorem exists. Then $T(S)$ is of the form

$T(S) \equiv \ldots; [\ldots$ **while** $b \wedge c$ **do** $\cdots$ **od**; $\ldots; b :=$ **true**; $\ldots \| \ldots; b :=$ **false**; $\ldots]$.

Consider now a computation of $T(S)$ that starts in a state σ where b is true and that gives preference to the first component as long as it is not terminated. This computation is finite since $T(S)$ always terminates. Thus, the first component eventually terminates. The only action $T(S)$ can subsequently take is to fully execute its second component. Thereafter the program $T(S)$ terminates in a state where b is false. But all fair computations of S starting in σ terminate in a state where b is true. Contradiction. □

The theorem points at the fundamental difference between nondeterministic and parallel programs: Nondeterministic programs have only one point of control that can easily be influenced by adding the scheduler on the top of the

do-od-statement; parallel programs have several points of control, one for each component. Thus when the scheduler blocks a component, its point of control has to be remembered. For this purpose we need the **await**-statements, even when transforming simple $L(\|)$-programs.

A solution. Thus our transformation will be of the form $T: L(\|) \to L(\|,$ **await**, ?). T should be *schematic* in the sense that, after performing steps (1) and (2) of Definition 4.3, we may encapsulate any sequential subprogram S' into an **await**-statement

await c **then** S' **end**.

Consider now an $L(\|)$-program

$$S \equiv S_0; [S_1 \| \cdots \| S_n].$$

Embedding FAIR into S is done as follows. The variables $z_1, \ldots, z_n$ of FAIR become auxiliary variables added to S. Since in $L(\|)$ enabledness means nontermination, the set E of enabled components is determined using the additional Boolean variables $\text{end}_1, \ldots, \text{end}_n$ satisfying $i \in E$ iff $\neg\, \text{end}_i$.

Checking a selection (E, i) in a run of S is done by enclosing in the ith component "sufficiently many" atomic statements A in **await**-statements

await SCH_i **then** UPDATE_i; A **end**.

But what are "sufficiently many"? According to Theorem 4.1 we have to ensure that each run of S is checked at the positions in some infinite check set. These positions will correspond to the execution of the atomic statements A just considered. In particular, we have to ensure that in every round a **while**-loop passes through such an atomic statement. This leads us to the following notion:

Definition 4.5. An *immediate atomic statement* of a loop **while** b **do** S' **od** is an atomic statement that occurs in S' but outside any further **while**-loop within S'. For example, in

while b **do while** c **do** $x := 1$ **od od**

the assignment $x := 1$ is an immediate atomic statement of the loop **while** c **do** $x := 1$ **od**, but not of **while** b **do** $\cdots$ **od** which, in fact, has no immediate atomic statement. Thanks to the following Padding Lemma such a missing immediate statement can be introduced without changing the semantics, for example, by replacing **while** b **do** S' **od** by **while** b **do** skip; S' **od**.

LEMMA 4.6 (Padding Lemma). *Consider a program $S \in L(\|)$. Let S^* result from S by replacing an occurrence of a substatement S_0 in S by "skip; S_0" or "S_0; skip." Then*

$$\mathcal{M}[\![S]\!] = \mathcal{M}[\![S^*]\!].$$

The same statement is true for all other types of programs and semantics considered in this paper.

Formally, the embedding of FAIR is done by the following transformation T_{fair}. For

$$S \equiv S_0; [S_1 \| \cdots \| S_n]$$

let $T_{\text{fair}}(S)$ result from S by

(1) prefixing S with the *initialization part*

$z_1 := ?; \ldots; z_n := ?;\ \text{end}_1 := \textbf{false}; \ldots; \text{end}_n := \textbf{false},$

(2) replacing in every **while**-loop the first immediate atomic statement A by

$\text{TEST}_i(A) \equiv \textbf{await}\ \text{SCH}_i\ \textbf{then}\ \text{UPDATE}_i; A\ \textbf{end}$

where, following FAIR, we have

$\text{SCH}_i \equiv z_i = \min\{z_k \mid \neg\, \text{end}_k\},$
$\text{UPDATE}_i \equiv z_i := ?;$
for all $j \in \{1, \ldots, n\} - \{i\}$ **do**
if $\neg\, \text{end}_j$ **then** $z_j := z_j - 1$ **fi**
od

and where $i \in \{1, \ldots, n\}$ is the index of the parallel component in which A occurs, and

(3) suffixing the ith component of S by the *termination part*

$\text{end}_i := \textbf{true}, \quad \text{for } i = 1, \ldots, n.$

Of course, the z_i's and end_i's are new variables not present in S.

THEOREM 4.7 (Embedding). *For every $L(\|)$-program S the equation*

$$\mathcal{M}_{fair}[\![S]\!] = \mathcal{M}[\![T_{fair}(S)]\!] \text{mod } Z$$

holds where Z is the set of auxiliary variables z_i and end_i in T_{fair}.

PROOF. Let S' result from $T_{\text{fair}}(S)$ by replacing the initial assignments $z_i := ?$, $\text{end}_i := \textbf{false}$, and the final assignments $\text{end}_i := \textbf{true}$ by "skip," and the new **await**-statements by the enclosed atomic statement A. Note that S' differs from S by some additional skip statements. Thus using the Padding Lemma 4.6, it suffices to show

$$\mathcal{M}_{\text{fair}}[\![S']\!] = \mathcal{M}[\![T_{\text{fair}}(S)]\!] \text{mod } Z. \tag{1}$$

The advantage of using S' instead of S is that the computations of S' and $T_{\text{fair}}(S)$ are "running step in step." This is needed in the last of the following equivalences, stated for an arbitrary interleaved computation ξ.

ξ is a fair computation of S'

iff ξ is a computation of S' with a fair, monotonic run (definition of fairness, Monotonicity Lemma 3.1),

iff ξ is a computation of S' with a run checked by the scheduler FAIR at the position in an arbitrary infinite check set (Theorem 4.1),

iff there exists a computation ξ^* of $T_{\text{fair}}(S)$ such that ξ is the restriction of ξ^* to the program parts and variables in S (construction of T_{fair}, definition of S', deadlock freedom of FAIR, and hence ξ^*).

Here *restriction* means that ξ is obtained from ξ^* by simplifying each configuration $\langle T_j, \tau_j \rangle$ in ξ^* as follows:

—in T_j every new **await**-statement is replaced by the enclosed atomic statement A and each remaining assignment to z_i and end_i by "skip,"

—in τ_j the variables z_i and end_i are reset to the values in the first state of ξ^*.

The above equivalences clearly imply (1), the desired result. □

Discussion. T_{fair} provides an example of transformational semantics: Fair parallelism is reduced to usual parallelism. The reduction is remarkable because it localizes the unbounded nondeterminism that fairness unavoidably introduces in random assignments (cf. Section 3). These assignments are part of a very general scheduling policy FAIR which enables the simulation of every fair run. Embedding FAIR into a parallel program S yields the transformed program $T_{\text{fair}}(S)$ which, no matter how its random assignments are then implemented, is guaranteed to generate only fair multiprocessor executions of the original program S. Thus FAIR and T_{fair} can be viewed as a template for an arbitrary implementation of fairness. Another advantage, its applicability in correctness proofs, will be explained later in Section 6.

Let us review some of the design decisions that went into T_{fair}. We insisted on transforming the *first* immediate atomic statement of every **while**-loop, but equally well we could have chosen *any other* immediate atomic statement. This does not affect the Embedding Theorem 4.7.

Taking a finer grain of interleaving as indicator for greater efficiency, we see that the transformed program $T_{\text{fair}}(S)$ is less efficient than the original version S. This is because in $T_{\text{fair}}(S)$ certain atomic statements A of S have been replaced by statements $\text{TEST}_i(A)$. We can improve the efficiency of T_{fair} by moving the original atomic statement A out of the **await**-statement. This yields a new transformation T_{fair^*} with

$\text{TEST}_i(A) \equiv$ **await** SCH_i **then** UPDATE_i **end**;
A.

Using the Padding Lemma 4.6, it is easy to see that the Embedding Theorem 4.7 remains valid for T_{fair^*}.

We can reduce the scope of the **await**-statement even further by taking out all the updates of the variables $z_1, \ldots, z_n$. This results in

$\text{TEST}_i(A) \equiv$ **wait** SCH_i;
UPDATE_i;
A

where **wait** B, for some Boolean expression B, abbreviates **await** B **then** skip **end**. Note that in the context of parallel composition the updates of $z_1, \ldots, z_n$ can now be delayed. Nevertheless, by a somewhat tedious analysis, it can be shown that the resulting transformation T^{**}_{fair} still satisfies

$$\mathcal{M}[\![T^{*}_{\text{fair}}(S)]\!] = \mathcal{M}[\![T^{**}_{\text{fair}}(S)]\!]$$

for every $L(\,\|\,)$-program S. Consequently, the Embedding Theorem 4.7 holds for T^{**}_{fair} as well.

5. TRANSFORMATIONAL SEMANTICS FOR WEAK AND STRONG FAIRNESS

We now extend the principle of transformational semantics to the full language $L(\,\|\,,$ **await**) in which weak and strong fairness are distinguished.

5.1 Schedulers

In the presence of **await**-statements, disabled components of parallel programs may become enabled again. Thus, runs of programs in $L(\|, \textbf{await})$ are not monotonic any more. Consequently, a scheduler enforcing weak or strong fairness must check the selections in a run more often than just once during every round through a **while**-loop.

Consider for example the program

$S^{**} \equiv$ [**await** c **then** $b_1 := \textbf{false}$ **end**
$\|$ **await** $\neg c$ **then** $b_2 := \textbf{false}$ **end**
$\|$ **while** $b_1 \vee b_2$ **do** $c := \neg c; c := \neg c$ **od**].

(1) (2)

Assuming strong fairness, S^{**} will terminate when starting in a state σ with $\sigma(b_1) = \sigma(b_2) = \sigma(c) = \textbf{true}$ because both its first and second components will be activated eventually. But a scheduler checking the condition c only at position 2 would find c always disabled and thus never activate the first component. Symmetrically, a scheduler checking c only at the loop entrance 1 would find $\neg c$ always disabled and thus never activate the second component. So a scheduler guaranteeing strongly fair runs of S^{**} should check the condition c both at 1 and 2.

In general, the check set $\mathscr{C}$ needs to be very dense: It must contain almost all positions j in which the selected component i_j changes the enabledness of the other components. Except for this change, we can reuse the scheduler FAIR of the previous section.

THEOREM 5.1 (FAIR Enforcing Strong Fairness). *Consider a run*

$$(E_0, i_0) \cdots (E_j, i_j)(E_{j+1}, i_{j+1}) \cdots$$

and an infinite check set $\mathscr{C}$ containing all, but finitely many, j such that

$$E_j - \{i_j\} \neq E_{j+1} - \{i_j\}.$$

Then the run is strongly fair iff it can be checked by FAIR at the positions in $\mathscr{C}$.

PROOF. The argument is similar to the proof of Theorem 4.1 and is hence only outlined.

If: Every run checked by FAIR is strongly fair. This is shown using the invariant INV of FAIR established in the proof of Theorem 4.1.

Only If: Every strongly fair run can be checked by FAIR; the values of the scheduling variables $z_1, \ldots, z_n$ in the sequence $\sigma_0 \cdots \sigma_j \cdots$ of scheduler states is given by

$$\sigma_j(z_i) = 1 + \text{card}\{l \in \mathscr{C} \mid j \leq l \leq m_{i,j} \wedge i \in E_l\}$$

for
$$i \in \{1, \ldots, n\}, \quad j \in \mathbb{N}_0 \quad \text{and}$$
$$m_{i,j} = \min\{m \in \mathscr{C} \mid j \leq m \wedge (i_m = i \vee \forall n \geq m : i \notin E_n)\}$$

The difference with the corresponding assignments in the proof of Theorem 4.1 is due to the possible nonmonotonicity of the run. Note that, as in

Theorem 4.1, the variables $z_1, \ldots, z_n$ take values ≥ 1. Thus $\sigma_j(z_i)$ counts the number of times the ith component will be enabled in a selection checked by the scheduler before its next (if any) activation. In particular, $\sigma_j(z_i)$ cares for the possible nonmonotonicity of the run. □

To model weak fairness, we cannot use the scheduler FAIR any more. The problem lies in the way FAIR updates its scheduling variables $z_1, \ldots, z_n$. When checking a selection (E, i) the scheduler FAIR decrements the variables z_j of all other enabled components $j \neq i$, but the variables z_j of disabled components $j \notin E$ are left unchanged. This is appropriate for strong but not for weak fairness in which a component should be *continuously* enabled. Here the variables z_j of disabled components should be reset.

This is achieved by changing the part UPDATE$_i$ of FAIR as follows:

```
UPDATE-W_i ≡ z_i := ?;
             for all j ∈ {1, ..., n} − {i} do
               if j ∈ E then z_j := z_j − 1
                 else z_j := ? fi
             od.
```

Let us call the resulting scheduler WFAIR.

THEOREM 5.2 (*WFAIR Enforcing Weak Fairness*). *Consider a run and a check set $\mathscr{C}$ as in Theorem* 5.1. *Then the run is weakly fair iff it can be checked by WFAIR at the positions in $\mathscr{C}$.*

PROOF. Analogous to that of Theorem 5.1. □

WFAIR also enforces fairness in monotonic runs; that is, Theorem 4.1 remains valid with WFAIR instead of FAIR. However, WFAIR is unnecessarily complicated for this purpose because resetting the variables z_j of disabled components is superfluous in monotonic runs.

Discussion. By Theorems 5.1 and 5.2, any scheduling policy for weak and strong fairness can be obtained from WFAIR and FAIR by implementing their random assignments (cf. Section 4.1). We discuss here two examples showing how realistic schedulers can be derived in this way.

The simplest method of enforcing weak fairness is by a *round robin scheduler* RORO which selects components clockwise, thereby skipping over momentarily disabled ones. We discussed RORO already in Section 4.1 in connection with fairness in monotonic runs. The implementation here is more complicated than in Section 4.1 because we now have to maintain the strictly clockwise scheduling policy of RORO for arbitrary runs. We take

```
        INIT ≡ z_1 := 1; z_2 := 2; ...; z_n := n;
        SCH_i ≡ z_i = min{z_k | k ∈ E},
UPDATE-W_i ≡ z_i := z_i − 1 + n;
             for all j ∈ {1, ..., n} − {i} do
               if j ∈ E then z_j := z_j − 1
               else if z_j < z_i + 1 − n then z_j := z_j − 1 + n
                 else z_j := z_j − 1 fi
               fi
             od.
```

This implementation ensures that at every selection via SCH_i the set of values stored in the scheduling variables $z_1, \ldots, z_n$ forms an interval

$$\{k + 1, k + 2, \ldots, k + n\}, \qquad (*)$$

for some $k \geq 0$. Initially, we simply have $k = 0$, and 1 is stored in z_1, 2 in $z_2, \ldots,$ and n in z_n. In general, the component i_0 with z_{i_0} containing the least value $k + 1$ is the current candidate for selection. We say "candidate" because i_0 need not be enabled. Selected via SCH_i is the component i which is enabled and comes closest after i_0 in the clockwise ordering. The update of the variables $z_1, \ldots, z_n$ ensures that the interval property $(*)$ is preserved (though possibly for a larger constant k).

By the interval property, the variables $z_1, \ldots, z_n$ can be transformed into one variable ranging over $\{1, \ldots, n\}$; this yields the most efficient implementation of RORO explained in Section 4.1 (cf. also [12, 27]).

Strong fairness cannot be enforced by an inexpensive round robin scheduling policy. As shown in [12], any strongly fair scheduler for n components needs at least $n!$ states. One way of organizing such a scheduler is by keeping the components in a queue [12, 27]. In each check the scheduler activates that enabled component which is earliest in the queue. This component is then placed at the end of the queue. Strong fairness is guaranteed since every enabled but not activated component advances one position in the queue. Let us call this scheduler QUEUE.

We show that the effect of QUEUE can be modelled by implementing the random assignments of our general scheduler FAIR in a specific way. We take

$$\begin{aligned}
&\mathrm{INIT} \equiv \textbf{for } i \in \{1, \ldots, n\} \textbf{ do } z_i := (i - 1) \cdot n \textbf{ od},\\
&\mathrm{SCH}_i \equiv z_i = \min\{z_k \mid k \in E\},\\
&\mathrm{UPDATE}_i \equiv z_i := n + \max\{z_1, \ldots, z_n\};\\
&\qquad\qquad \textbf{for } j \in \{1, \ldots, n\} - \{i\} \textbf{ do}\\
&\qquad\qquad\quad \textbf{if } j \in E \textbf{ then } z_j := z_j - 1 \textbf{ fi}\\
&\qquad\qquad \textbf{od}.
\end{aligned}$$

The idea is that in the QUEUE component i comes before component j iff $z_i < z_j$ holds in the above implementation. Since FAIR leaves the variables z_j of disabled components j unchanged and decrements those of enabled but not activated ones, some care had to be taken in the implementation of the random assignments of FAIR in order to prevent any "overtaking" of components within the queue. More precisely, the order "component i before component j," represented by $z_i < z_j$, should be preserved as long as neither i nor j is activated. That is why initially and in every update we keep a difference of n between the new value of z_i and all previous values. This difference is sufficient because a component that is enabled n times is selected at least once.

5.2 Transformations

We obtain transformations T_{wfair} and T_{sfair} for weak and strong fairness by embedding the schedulers WFAIR and FAIR into the programs in $L(\|, \textbf{await})$. These transformations are $\|$-preserving and schematic as T_{fair} for $L(\|)$ but produce more complicated programs first, because determining the enabledness of components is more elaborate and second, because selections need to be checked more often because of Theorems 5.1 and 5.2.

Given a program $S \equiv S_0; [S_1 \parallel \cdots \parallel S_n]$, both transformations will add the following auxiliary variables to S: z_i, end_i, pc_i, $i = 1, \ldots, n$. The z_i's and end_i's are used as in T_{fair}. The pc_i's are a restricted form of program counters that indicate when the component S_i is in front of an **await**-statement and, if so, in front of which one. To this end, we assign to every occurrence of an **await**-statement in S_i a unique number $l \geq 1$ as label. Let L_i denote the set of all these labels for S_i, and B_l denote the Boolean expression of the **await**-statement labeled by l. We introduce the abbreviation

$$\text{enabled}_i \equiv \neg\text{end}_i \wedge \bigwedge_{l \in L_i} (pc_i = l \rightarrow B_l),$$

for $i = 1, \ldots, n$. In the transformed program enabled_i will evaluate to true iff the ith component of S is indeed enabled.

With these conventions the transformation

T_{sfair}: $L(\parallel, \textbf{await}) \rightarrow L(\parallel, \textbf{await}, ?)$

for strong fairness is as follows: Given a program

$S \equiv S_0; [S_1 \parallel \cdots \parallel S_n]$

in $L(\parallel, \textbf{await})$, let $T_{\text{sfair}}(S)$ result from S by performing the following:

(1) prefixing S with **for all** $i \in \{1, \ldots, n\}$ **do** $z_i := ?$; $\text{end}_i :=$ **false**; $pc_i := 0$ **od**;

(2) replacing every substatement **await** B_l **then** S' **end** with $l \in L_i$ in the ith component of S by

$pc_i := l$; **await** B_l **then** S'; $pc_i := 0$ **end**,

for $i = 1, \ldots, n$;

(3) replacing *every* immediate atomic statement A in a **while**-loop of the ith component of S by

$\text{TEST}_i(A) \equiv$ **await** SCH_i **then** UPDATE_i; A **end**

where, following FAIR, we have

$$\begin{aligned} \text{SCH}_i &\equiv z_i = \min\{z_k \mid \text{enabled}_k\}, \\ \text{UPDATE}_i &\equiv z_i := ?; \\ &\quad \textbf{for all } j \in \{1, \ldots, n\} - \{i\} \textbf{ do} \\ &\quad\quad \textbf{if } \text{enabled}_j \textbf{ then } z_j := z_j - 1 \textbf{ fi} \\ &\quad \textbf{od} \end{aligned}$$

for $i = 1, \ldots, n$. If A is already an **await**-statement, we "amalgamate" its Boolean expression with SCH_i to avoid nested **await's** which are disallowed in our syntax (cf. Section 2). As in Section 4.2, we stipulate here that every **while**-loop has an immediate atomic statement; if not we add one by applying the Padding Lemma 4.6.

(4) suffixing the ith component of S by

$\text{end}_i :=$ **true**, for $i = 1, \ldots, n$.

For weak fairness the transformation

T_{wfair}: $L(\parallel, \textbf{await}) \rightarrow L(\parallel, \textbf{await}, ?)$

is defined as T_{sfair}, but with UPDATE-W_i instead of UPDATE_i.

THEOREM 5.3 (Embedding). *For every program S in $L(\|, \textbf{await})$ the equations*

$$\mathcal{M}_{sfair}[\![S]\!] = \mathcal{M}[\![T_{sfair}(S)]\!] mod\ Z$$

and

$$\mathcal{M}_{wfair}[\![S]\!] = \mathcal{M}[\![T_{wfair}(S)]\!] mod\ Z$$

hold where Z is the set of auxiliary variables z_i, end_i, and pc_i.

PROOF. By a straightforward variation of the proof of the Embedding Theorem 4.7, now, of course, referring to the new Theorems 5.1 and 5.2 about FAIR and WFAIR. In contrast to Theorem 4.7, computations of S may now deadlock owing to the presence of **await**-statements. These deadlocks occur also in the computations ξ^* of the transformed programs $T_{\text{sfair}}(S)$ and $T_{\text{wfair}}(S)$. But since FAIR and, hence, WFAIR are deadlock free (cf. Section 4.1), no additional deadlock is possible in ξ^*. □

Discussion. The main difference between the new transformations, T_{wfair} and T_{sfair}, and T_{fair} is that now *every* immediate atomic statement A is enclosed in an **await**-statement $\text{TEST}_i(A)$. This seriously reduces the efficiency of the transformed programs. But is this reduction really unavoidable?

Looking at Theorems 5.1 and 5.2, about the schedulers FAIR and WFAIR, we conclude that activations of components that leave the enabledness of all other components unchanged need not be checked. In T_{wfair} and T_{sfair} we exploit this fact only for the evaluation of Boolean expressions. By exploiting it for atomic statements as well, we can improve the efficiency of the transformations.

Formally, we refine Step 3 of $T_{\text{wfair}}(S)$ and $T_{\text{sfair}}(S)$ by enclosing in each **while**-loop of the ith component of S at least one immediate atomic statement A in a test part $\text{TEST}_i(A)$ and additionally only those immediate atomic statements A that *change a variable which is referenced in the Boolean expression B of some statement* **await** B **then** S' **end** *in the original program S*. This refinement is particularly useful for programs with limited interaction among the parallel components; an example will be studied in the next section.

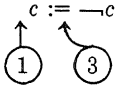
For the previous transformations T_{fair} we could improve efficiency by taking the atomic statement A out of the **await**-statement $\text{TEST}_i(A)$. Is this possible also for T_{wfair} and T_{sfair}? For strong fairness the answer is "no." Suppose we change T_{sfair} to a transformation T^*_{fair} by putting

$\text{TEST}_i(A) \equiv$ **await** SCH_i **then** UPDATE_i **end**;
A.

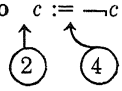
Then the resulting program $T^*_{\text{sfair}}(S)$ may fail to recognize that a component of S is infinitely often enabled.

For example, take the program

$S \equiv c :=$ **false**; [**while** b **do** $c := \neg c$ **od**



‖ **while** b **do** $c := \neg c$ **od**



await c **then** $b :=$ **false end**].

$T^*_{\text{sfair}}(S)$ would check the truth value of c at the positions 1 and 2 before executing the assignments 3 and 4. Thus it admits an infinite computation ξ which periodically activates 1, 2, 3, 4.

Since initially c evaluates to **false**, the checks at 1 and 2 will never find that c is **true** in between the execution of 3 and 4. So ξ is not strongly fair because the third component of S is infinitely often enabled but never activated. Consequently, $T^*_{\text{sfair}}(S)$ is incorrect.

For weak fairness, the above counterexample S does not apply because activation of a component is enforced by continuous enabledness. Indeed, the Embedding Theorem 5.3 remains valid for the transformations T^*_{wfair} obtained by taking

$$\text{TEST}_i(A) \equiv \underset{A}{\textbf{await SCH}_i \textbf{ then UPDATE-W}_i \textbf{ end}};$$

and T^{**}_{wfair} obtained by taking

$$\text{TEST}_i(A) \equiv \begin{array}{l}\textbf{wait SCH}_i;\\ \text{UPDATE-W}_i;\\ A\end{array}$$

as in the case of the transformation T_{fair}.

6. APPLICATIONS TO PROGRAM CORRECTNESS

We presented transformations that reduce fair parallelism semantics to the usual parallelism semantics. These transformations shed light on the assumption of fairness because they link the fairness policies with the schedulers that implement them. But since the transformations are structure preserving, they also provide a basis for syntax-directed correctness proofs of parallel programs executed under fairness assumptions.

The idea is to use the equivalence

$$\models_{\text{fair}}\{p\}S\{q\} \quad \text{iff} \quad \models\{p\}T_{\text{fair}}(S)\{q\} \tag{1}$$

and the corresponding ones for weak and strong fairness that follow immediately from the Embedding Theorems 4.8 and 5.3. $\{p\}S\{q\}$ and $\{p\}T_{\text{fair}}(S)\{q\}$ are the usual Hoare-style correctness formulas with precondition p and postcondition q. $\models_{\text{fair}}$ expresses total correctness under the assumption of fairness, and $\models$ expresses total correctness without any assumption. As usual, *total correctness* of parallel programs encompasses

—partial correctness,
—divergence freedom,
—deadlock freedom.

Thus to prove total correctness of S under the assumption of fairness, it suffices to prove total correctness of $T_{\text{fair}}(S)$ in the usual sense. This can be done with standard proof methods for parallel programs, extended by rules for the random assignments in $T_{\text{fair}}(S)$.

For partial correctness and divergence freedom we use an extension of the Owicki–Gries approach [24]; deadlock freedom will be treated separately. We assume familiarity with [24] but recall briefly the main ideas. In [24] a correctness proof of a parallel program S proceeds in two steps. First, one has to find

appropriate correctness proofs for the sequential components of S. This is done using the proof rules of [24] extended by the following ones taken from [3] and [16]:

Random Assignment Axiom.

$$\{p\}z := ?\{p \wedge z \geq 0\}$$

where z is not free in p, and the

Extended While Rule

$$\frac{\begin{array}{l}\{p \wedge b\}S\{p\},\\ \{p \wedge b \wedge t = \alpha\}S\{t < \alpha\},\\ p \wedge b \rightarrow t > 0\end{array}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ \{p \wedge \neg b\}}$$

where p is an assertion (called the *loop invariant*), t is an expression (called the *loop variant* or *bound function*), and α is a variable not appearing in t, b, or S.

The last two premises of the rule guarantee divergence freedom. Here however, owing to the presence of random assignments, it is, in general, not sufficient to let t be an integer expression and α an integer variable. Instead, we shall need expressions involving infinite ordinals and variables ranging over infinite ordinals as in [1], [3], [7], and [21] (for details see the sample S^*_{zero} below).

The second step combines the correctness proofs for the components using the

Parallel Composition Rule.

$$\frac{\text{The proofs of } \{p_1\}S_1\{q_1\}, \ldots, \{p_n\}S_n\{q_n\} \text{ are interference-free}}{\{p_1 \wedge \cdots \wedge p_n\}[S_1 \| \cdots \| S_n]\{q_1 \wedge \cdots \wedge q_n\}}$$

of [24]. In its premise this rule checks whether the correctness proofs for the components fit together. This is done using the test of *interference freedom.* Correctness proofs for sequential components are interference free if

(i) the pre- and postconditions, in particular the loop invariants, used in one proof cannot be invalidated by the execution of an atomic statement of another component, and

(ii) the loop variants used in one proof cannot be increased by an activation of an atomic statement of another component. That is, for each variant t and each atomic statement A with the precondition p from another proof the correctness formula

$$\{p \wedge t = \alpha\} \quad A\{t \leq \alpha\}$$

holds, where α is a variable ranging over ordinals and not appearing in t or A.

In case of (i) and (ii) we talk of *interference-free* invariants and variants.

Uniform $L(\|)$-Programs. Let us now explain our approach of combining program transformations and the Owicki–Gries method in more detail, first for $L(\|)$. The advantage of $L(\|)$-programs is that they cannot deadlock, whether we assume fairness or not. So, proving total correctness under fairness reduces

to proving partial correctness and divergence freedom under fairness. This can be done using the transformation T_{fair} and the equivalence (1) above.

However, we will simplify T_{fair} here by

(i) replacing the scheduling condition SCH_i by $z_1, \ldots, z_n \geq 1$, and
(ii) dropping the termination variables end_i.

These changes yield a transformation $T_{\text{fair}+\Delta}$, which behaves as T_{fair} except that it can deadlock. Of course, such a transformation should be rejected as an implementation of fairness, but it turns out to be useful for proving correctness since it leads to simpler loop invariants and bound functions.

Because of (ii) the transformation $T_{\text{fair}+\Delta}$ will work only for a subclass of $L(\|)$-programs, which we now introduce.

Definition 6.1. An $L(\|)$-program $S \equiv [S_1 \| \cdots \| S_n]$ is *uniform* if each of its components S_i, $i = 1, \ldots, n$, is of the form

$S_i \equiv$ **while** B **do** T_i **od**

for some uniform Boolean condition B and arbitrary loop-free statement T_i.

For example, the program S_{zero} of the introduction is uniform. For uniform programs $S \equiv S_0; [S_1 \| \cdots \| S_n]$ let $T_{\text{fair}+\Delta}(S)$ result from S by

(1) prefixing S with $z_1 := ? ; \cdots ; z_n := ?$, and
(2) replacing in every **while**-loop of S the first immediate atomic statement A by

$\text{TEST}_i(A) =$ **await** $z_1, \ldots, z_n \geq 1$ **then**
$z_i := ? ;$
for all $j \in \{1, \ldots, n\} - \{i\}$ **do** $z_j := z_j - 1$ **od**; A
end

where $i \in \{1, \ldots, n\}$ is the index of the component of S in which A occurs. As before, the z_i's are new variables not present in S.

Let $\mathcal{M}_{-\Delta}$ be a variant of the asynchronous parallelism semantics $\mathcal{M}$ that *ignores deadlocks*, that is, with

$$\mathcal{M}_{-\Delta}[\![S]\!](\sigma) = \mathcal{M}[\![S]\!](\sigma) - \{\Delta\}.$$

Then we can state:

THEOREM 6.2. *For uniform $L(\|)$-programs S the equation*

$$\mathcal{M}_{fair}[\![S]\!](\sigma) = \mathcal{M}_{-\Delta}[\![T_{fair+\Delta}(S)]\!] mod\ Z$$

with $Z = \{z_1, \ldots, z_n\}$ holds.

PROOF. Similar to the proofs of Theorems 4.1 and 4.7 but deals only with terminating and diverging computations. Uniformity of S guarantees that in a diverging fair computation *every* component of S gets activated infinitely often. Hence a corresponding diverging computation of $T_{\text{fair}+\Delta}(S)$ can be constructed without the help of the termination variables end_i. □

For nonuniform programs the theorem is wrong. Consider, for example,

$S \equiv$ [**while true do** skip **od** $\|$ skip].

Whereas S diverges, even under the assumption of fairness, $T_{\text{fair}+\Delta}(S)$ can only deadlock. To preserve divergence of S, the original transformation T_{fair} uses the termination variable end_2.

An immediate consequence of Theorem 6.2 is the equivalence

$$\vDash_{\text{fair}}\{p\}S\{q\} \quad \text{iff} \quad \vDash_{-\Delta}\{p\}T_{\text{fair}+\Delta}(S)\{q\} \tag{1'}$$

where $\vDash_{-\Delta}$ expresses total correctness *modulo deadlocks*, that is, only partial correctness and divergence freedom. This is exactly the type of correctness that can be proved by the extended Owicki–Gries method explained above.

Zero Searching. As an example of a uniform $L(\|)$-program we consider now the following version S^*_{zero} of the introductory zero searching program:

$S^*_{\text{zero}} \equiv$ [**while** $f(x) \neq 0 \wedge f(y) \neq 0$ **do** $x := x - 1$ **od**
$\|$ **while** $f(x) \neq 0 \wedge f(y) \neq 0$ **do** $y := y + 1$ **od**].

Let S^*_{zero} start in a state satisfying the condition

$$\exists u: f(u) = 0 \wedge x = y.$$

We claim that under the fair parallelism semantics $\mathcal{M}_{\text{fair}}$ the program S^*_{zero} is certain to terminate in a state satisfying

$$f(x) = 0 \vee f(y) = 0.$$

In terms of Hoare-style correctness formulas the claim is

$$\vDash_{\text{fair}}\{\exists u: f(u) = 0 \wedge x = y\}S^*_{\text{zero}}\{f(x) = 0 \vee f(y) = 0\}. \tag{2}$$

Using the equivalence (1') this claim is equivalent to

$$\vDash_{-\Delta}\{\exists u: f(u) = 0 \wedge x = y\}T\{f(y) = 0 \vee f(y) = 0\} \tag{2'}$$

where the transformed program $T \equiv T_{\text{fair}+\Delta}(S^*_{\text{zero}})$ appearing in claim (2') is

$T \equiv z_1 := ?;\ \ z_2 := ?;$
[**while** $f(x) \neq 0 \wedge f(y) \neq 0$ **do**
await $z_1, z_2 \geq 1$ **then**
$z_1 := ?;\ \ z_2 := z_2 - 1;$
$x := x - 1$
end
od] $\}\ T_1$

[**while** $f(x) \neq 0 \wedge f(y) \neq 0$ **do**
await $z_1, z_2 \geq 1$ **then**
$z_1 := z_1 - 1;\ \ z_2 := ?;$
$y := y + 1$
end
od] $\}\ T_2$

To prove

$$\{\exists u: f(u) = 0 \wedge x = y\}T\{f(x) = 0 \vee f(y) = 0\}$$

we split the precondition into two subcases:

(a) $f(u) = 0 \wedge u \leq x = y$
(b) $f(u) = 0 \wedge u \geq x = y$.

Let us study subcase (a) where a zero u can be found by activating the first component T_1 of T sufficiently often. For (a) we wish to prove

$$\{f(u) = 0 \wedge u \leq x = y\}T\{f(x) = 0 \vee f(y) = 0\}.$$

We have to find interference-free loop invariants p_1 and p_2 and loop variants t_1 and t_2 for the sequential components T_1 and T_2 of T. It is clear that T_1 terminates owing to the fact that x gets decremented and $f(u) = 0 \wedge u \leq x$ holds invariantly. Thus $t_1 = x - u$ seems to be an obvious choice for the variant T_1.

Summarizing, we choose

$$p_1 \equiv f(u) = 0 \wedge u \leq x$$

and

$$t_1 \equiv x - u.$$

Clearly p_1 and t_1 satisfy the premise of the Extended While Rule when applied to T_1. This yields

$$\{p_1\}T_1\{p_1 \wedge (f(x) = 0 \vee f(y) = 0)\}.$$

Moreover, p_1 and t_1 are interference free with respect to the second component T_2 because neither u nor x is changed by T_2.

The situation is a bit more complicated with the loop invariant p_2 and the loop variant t_2 of T_2. If T_2 were executed in isolation, z_1 would be an appropriate choice for t_2. Unfortunately, this condition of T_2 is not interference free with respect to T_1: An activation of T_1 can increase t_2 by executing $z_1 := ?$. However, indivisibly coupled with every reset of z_1 by T_1 is a decrease of the variable x by the assignment $x := x - 1$. This observation suggests the following loop invariant and loop variant for T_2:

$$p_2 \equiv f(u) = 0 \wedge u \leq x \wedge z_1 \geq 0$$

and

$$t_2 \equiv (x - u) \cdot \omega + z_1.$$

Here the first infinite ordinal ω is used, and

$$(x - u) \cdot \omega + z_1$$

corresponds to the lexicographical ordering of pairs $\langle x - u, z_1 \rangle$. Now p_2 and t_2 are indeed interference free with respect to T_1 because an increase of z_1 is compensated by a decrease of $x - u$ in $t_2 = (x - u) \cdot \omega + z_1$. Again, it is easy to see that p_2 and t_2 satisfy the premise of the Extended While Rule applied to T_2; that is, we prove

$$\{p_2\}T_2\{p_2 \wedge (f(x) = 0 \vee f(y) = 0)\}.$$

Using (essentially) the Parallel Composition Rule we arrive at

$$\{f(u) = 0 \wedge u \leq x = y\}T\{f(x) = 0 \vee f(y) = 0)\}.$$

Subcase (b) above is treated analogously. This completes the correctness proof of claim (2′) and hence of claim (1). □

$L(\|, Await)$-Programs. In the case of general $L(\|, \textbf{await})$ programs $S \equiv S_0; [S_1 \| \cdots \| S_n]$ we can adopt a similar approach as long as we do not intend to prove deadlock freedom. We demonstrate this here only for the case of strong fairness, but the analogous approach works for weak fairness.

The appropriate transformation $T_{\text{sfair}+\Delta}$ is obtained from T_{sfair}, simplifying the condition SCH_i in $\text{TEST}_i(A)$ to $z_1, \ldots, z_n \geq 1$. Clearly, the resulting program $T_{\text{sfair}+\Delta}(S)$ may deadlock but we still obtain

THEOREM 6.3. *For all $L(\|, \textbf{Await})$-programs $S \equiv S_0; [S_1 \| \cdots \| S_n]$ the equation*

$$\mathcal{M}_{sfair-\Delta}[\![S]\!] = \mathcal{M}_{-\Delta}[\![T_{sfair+\Delta}(S)]\!] mod\ Z$$

with $Z = \{z_1, \ldots, z_n, end_1, \ldots, end_n\}$ holds. As expected, $\mathcal{M}_{sfair-\Delta}$ stands for the version of $\mathcal{M}_{sfair}$ which ignores deadlocks.

PROOF. Following Theorems 5.1 and 5.3, every strongly fair computation of S can be modeled by $T_{\text{sfair}}(S)$ with the scheduling variables $z_1, \ldots, z_n$ taking values ≥ 1. Hence, it can be modeled by $T_{\text{sfair}+\Delta}(S)$. In contrast to Theorem 5.3, computations of $T_{\text{sfair}+\Delta}(S)$ can lead to deadlock even if S is deadlock free. However, this does not matter here since all deadlocks are ignored. □

As before, we are interested in the following consequence of Theorem 6.3:

$$\vDash_{\text{sfair}-\Delta}\{p\}S\{q\} \quad \text{iff} \quad \vDash_{-\Delta}\{p\}T_{\text{sfair}+\Delta}(S)\{q\} \tag{3}$$

where $\vDash_{\text{sfair}-\Delta}$ is defined analogously to $\vDash_{-\Delta}$ and thus expresses only the partial correctness and finiteness of strongly fair computations. We make use of this equivalence in the following example:

Mutual Exclusion—Eventual Access. Consider the following program from $L(\|, \textbf{await})$:

$S \equiv b := \textbf{true}; [S_1 \| S_2]$

where for $i = 1, 2$

$S_i \equiv$ **while true do**
R$_i$;
P(b);
CS$_i$;
V(b)
od.

Here P(b) and V(b) are the following abbreviations:

P(b) ≡ **await** b **then** $b :=$ **false end**,
V(b) ≡ $b :=$ **true.**

We assume that the not further specified parts R_i and CS_i are loop- and **await**-free sequential programs that do not modify the variable b.

S can be viewed as a solution to the mutual exclusion problem using a binary semaphore b. CS_1 and CS_2 are the critical sections, and R_1 and R_2 are the noncritical sections. P(b) and V(b) model Dijkstra's semaphore operations.

In fact, the following can be proved using the standard approach of [24]:

CLAIM 1. (Mutually Exclusive Access). *Consider a computation ξ of S. In every configuration of ξ the control in at most one of the components is within its critical section CS_i.*

We now prove the following additional property of the program S:

CLAIM 2. (Eventual Access). *In every strongly fair computation of S the control in the first component S_1 eventually enters its critical section CS_1. More formally, in every strongly fair computation*

$$\xi = \langle S, \sigma \rangle \rightarrow \langle T_1, \sigma_1 \rangle \rightarrow \cdots \tag{4}$$

some T_i is of the form

$$[\mathrm{CS}_1; \mathrm{V(b)}; S_1 \parallel S_2'] \tag{5}$$

for some S_2'. By symmetry the same statement holds for the second component.

How can Claim 2 be formalized in the proof-theoretic framework used in this paper? We express it as a correctness formula under the interpretation $\models_{\mathrm{sfair}-\Delta}$.

Note: Claim 2 holds iff

$$\models_{\mathrm{sfair}-\Delta}\{b\}[\mathrm{R}_1; \mathrm{P(b)} \parallel S_2]\{\mathbf{true}\}. \tag{6}$$

PROOF. It suffices to show that Claim 2 holds iff every strongly fair computation of the program $S' \equiv b := \mathbf{true}$; $[\mathrm{R}_1; \mathrm{P(b)} \parallel S_2]$ is finite, that is, terminating or deadlocking.

Consider a strongly fair computation η of S'. It can be naturally modified to a strongly fair computation ξ of S of the form (4). By Claim 2 some T_i is of the form (5). By Claim 1 in the configuration $\langle T_i, \sigma_i \rangle$ the control in the second component is outside of CS_2 and b is **false**. This means that in η the first component of S' eventually terminates in a configuration in which the control in the second component is outside of CS_2 and b is **false**. By the assumption the execution of R_2 eventually terminates and, because $\neg b$ holds, a deadlock arises. Thus η is finite.

Consider a strongly fair computation ξ of S of the form (4). It can be naturally modified to a strongly fair computation η of S'. η is finite, which, by the form of S_2, implies that η terminates in deadlock. By the form of S', deadlock can arise only when the first component of S' terminates. But this means that in ξ some T_i is of the form (5). □

Now, to prove (6) we apply the deadlocking transformation $T_{\mathrm{strong}+\Delta}$ to the program given in (6). By (3) the formula (6) is equivalent to

$$\models_{-\Delta}\{b\}T\{\mathbf{true}\} \tag{7}$$

where T is $T_{\mathrm{sfair}+\Delta}([\mathrm{R}_1; \mathrm{P(b)} \parallel S_2])$. In full expansion we have

$T \equiv z_1 := ?; z_2 := ?; \mathrm{end}_1 := \mathbf{false}; \mathrm{end}_2 := \mathbf{false}; pc_1 := 0; pc_2 := 0;$

```
    [R1; P1(b);                 }
     end1 := true               }   T1
     || while true do           )
          R2;                   )
          P2(b);                )
          CS2;                  )   T2
          V2(b)                 )
        od;                     )
        end2 := true]           )
```

with

$P_1(b) \equiv pc_1 := 1;$ **await** b **then** $b :=$ **false**; $pc_1 := 0$ **end**,
$P_2(b) \equiv pc_2 := 1;$
await $b \wedge z_1, z_2 \geq 1$ **then**
$z_2 :=$?; **if** enabled$_1$ **then** $z_1 := z_1 - 1$ **fi**;
$b :=$ **false**; $pc_2 := 0$
end,
$V_2(b) \equiv$ **await** $z_1, z_2 \geq 1$ **then**
$z_2 :=$?; **if** enabled$_1$ **then** $z_1 := z_1 - 1$ **fi**;
$b :=$ **true**
end

and

$\text{enabled}_1 \equiv \neg\text{end}_1 \wedge (pc_1 = 1 \rightarrow b)$.

Note that we make use of the discussion in Section 5.2 and transform only those atomic statements of the original **while**-loop which change the Boolean variable b, namely, $P(b)$ and $V(b)$, now yielding $P_2(b)$ and $V_2(b)$ in T_2.

We prove the correctness formula (7) again with the extended Owicki–Gries method. Clearly, the initialization in T yields as a postcondition

$$r \equiv b \wedge \bigwedge_{i=1,2} z_i \geq 0 \wedge \neg\text{end}_i \wedge pc_i = 0.$$

It remains to show

$$\{r\}[T_1 \parallel T_2]\{\textbf{true}\}. \tag{8}$$

This is fairly simple. For the first component T_1 we choose the proof with all assertions **true**, but we put the assertion $\neg b$ before and after its last assignment $\text{end}_1 :=$ **true**.

For the second component, T_2, we take as loop invariant,

$$p_2 \equiv z_1 \geq 0 \wedge (b \rightarrow \text{enabled}_1)$$

and as loop variant,

$$t_2 \equiv z_1 + 1.$$

Clearly, p_2 is kept invariant by the loop in T_2. Also, we have for each statement A inside R_2, CS_2, and $V_2(b)$,

$$\{p_2 \wedge t = \alpha\}A\{t \leq \alpha\}$$

as the only action possibly affecting z_1 is the assignment $z_1 := z_1 - 1$ within $V_2(b)$. Moreover, for $P_2(b)$, we have

$$\{p_2 \wedge t = \alpha\}P_2(b)\{t < \alpha\},$$

since p_2 guarantees that the assignment $z_1 := z_1 - 1$ within $P_2(b)$ is indeed executed. Thus t gets decreased in each complete round through the loop. Finally, we have

$$p_2 \rightarrow t_2 > 0.$$

With the Extended While Rule, we get

$$\{p_2\}T_2\{\textbf{true}\}.$$

We now check the interference freedom of these proofs. For the first component T_1 we have to show that the assertion $\neg b$ before and after the assignment $\text{end}_1 := \textbf{true}$ is not affected by the atomic statements in T_2. This can be done by strengthening the above proofs through appropriate auxiliary variables in the sense of [24]. Supplying full details would in fact amount to the axiomatic proof of Claim 1; we omit this standard application of the Owicki–Gries approach here.

For the second component, T_2, first consider the loop invariant p_2. Its first conjunct, $z_1 \geq 0$, is always preserved. Now, the only action affecting the second conjunct $b \rightarrow \text{enabled}_1$ is the assignment $\text{end}_1 := \textbf{true}$ in T_1. However, its precondition is $\neg b$, so after its execution $\neg b$ still holds, and consequently the conjunct $b \rightarrow \text{enabled}_1$ as well. Consider now the loop variant t_2. Obviously, no action within T_1 affects t_2.

Using the Parallel Composition Rule and the implication $r \rightarrow p_2$, we get (8). This finishes the proof of Claim 2. □

Discussion. We conclude with some comments on the above proofs. First, note that the auxiliary variable z_1 in the transformed programs plays the role of a "helpful variable" when formulating the loop variant t_2 of (cf. [15], Chap. 2). Without z_1 we cannot find an appropriate loop variant that is decremented with every execution of the loop body of T_2.

Second, to prove fair total correctness of an $L(\|)$-program S it suffices to prove total correctness modulo deadlock of the transformed program T. Therefore it suffices also to use simple deadlocking transformations like $T_{\text{fair}+\Delta}$. For describing schedulers we are, of course, advised to use deadlock free transformations only. We interpret this observation as follows: In proofs of program correctness we need not worry about the exact course of a computation but rather more abstractly about its results. This inherent abstraction in program proving allows us to employ transformations that model the program behavior in an imprecise manner.

Third, note that in the above correctness proof we did not reason about the original program S but its transformed version T. This should be contrasted with the approach taken in [1] and [5] to reason about fairness in nondeterministic **do-od**-programs. There we also started with transformations realizing the fairness assumptions, but in a second step when we developed proof rules dealing with fairness we were able to "absorb" the transformations into the assertions of existing rules. Thus the resulting proof rules for fairness could be applied directly to the original **do-od**-programs.

For parallel programs the idea of absorption does not work properly because of the test of interference freedom: When applied to the transformed program it has to deal also with all assignments affecting the auxiliary variables z inside the added await-statements. So even if these await-statements were absorbed into the assertions of the standard proof rules for the sequential components of parallel programs, they would reappear in the final test of interference freedom. We, therefore, propose to apply the transformations explicitly as a part of the correctness proofs.

7. CONCLUDING REMARKS

We briefly discuss alternative approaches to semantics and correctness of parallel programs under fairness assumptions and report on further developments.

Process Semantics. For simplicity, we introduced in Section 2 a semantics $\mathcal{M}[\![S]\!]$ that stresses termination of parallel programs S. Therefore all infinite computations of S are identified with the divergence symbol $\perp$. Equally well we might consider a more discriminating *process semantics* $\Pi[\![S]\!]$ which preserves the "essence" of infinite computations.

The basic idea is that $\Pi[\![S]\!]$ records for each computation of S the finite or infinite sequence of states. In case of deadlock, Δ is added as a final symbol. More precisely, we vary this idea in two respects.

First, we shall be interested only in the values of a given set X of variables, that is, we consider only the restrictions $\sigma \restriction X$ of states σ to these variables. The set X appears as a parameter of the process semantics: $\Pi[\![S, X]\!]$. This is convenient when we wish to ignore changes of auxiliary variables used in transformations.

Second, we adopt a proposal of [6] and require that our process semantics be insensitive to *finite* stuttering but be able to recognize *infinite* stuttering. Following Lamport, *stuttering* is the repetition of identical states [20]. Insensitivity to finite stuttering allows, for example, the insertion of skip statements into parallel programs without changing the semantics. Thus the Padding Lemma 4.6, used in the correctness proof of our transformations, remains valid. On the other hand, recognition of infinite stuttering is needed to distinguish between termination and divergence.

Formally, for any finite or infinite computation

$$\xi\colon \langle S_1, \sigma_1\rangle \to \cdots \to \langle S_j, \sigma_j\rangle \cdots$$

and any set X of variables, let filter(ξ, X) denote the subsequence of

$$(\sigma_1 \restriction X) \cdots (\sigma_j \restriction X) \cdots$$

obtained by removing all finite repetitions but keeping all infinite ones. Further on, let Σ^* (respectively, Σ^ω) denote the set of all finite (respectively, infinite) sequences of states in Σ. Then we define for $S \in L(\|, \textbf{await}, ?)$ and $X \subseteq \mathrm{Var}$ the *process semantics*

$$\Pi[\![S, X]\!]\colon \Sigma \to \mathcal{P}(\Sigma^* \cup \Sigma^* \cdot \{\Delta\} \cup \Sigma^\omega)$$

by

$$\begin{aligned}\Pi[\![S, X]\!] = \{&\mathrm{filter}(\xi, X) \mid \xi \text{ is a terminating or}\\ &\text{infinite computation of } S \text{ starting in } \sigma\}\\ \cup \{&\mathrm{filter}(\xi, X) \cdot \Delta \mid \xi \text{ is deadlocking computation}\\ &\text{of } S \text{ starting in } \sigma\}.\end{aligned}$$

For example, a state σ with $\sigma(x) = 0$ yields

$$\Pi[\![\textbf{while true do } \mathrm{skip};\ x := x + 1\ \textbf{od}, \mathrm{Var}]\!](\sigma) = \{\sigma\sigma[1/x]\sigma[2/x] \cdots\}.$$

For fairness assumptions $f \in$ {fair, wfair, and sfair}, we define $\Pi_f[\![S, X]\!]$ analogously to $\mathscr{M}_f[\![S]\!]$. For example, the *fair process semantics* $\Pi_{\text{fair}}[\![S, X]\!]$ considers only fair computations ξ. Our transformations T_{fair}, T_{wfair}, and T_{sfair} remain correct under the process semantics.

THEOREM 7.1 (Embedding: Process Semantics). *For every program $S \in L(\|)$ respectively $S \in L(\|,$ **await**, ?) and every fairness assumption $f \in$ {fair, wfair, sfair} the equation*

$$\Pi_f[\![S, Var(S)]\!] = \Pi[\![T_f(S), Var(S)]\!]$$

holds where $Var(S)$ is the set of variables in S.

PROOF. By inspection of the proofs of the previous Embedding Theorem 4.8 and 5.3. □

Recall that for T_{fair} and T_{wfair} we also discussed versions T^*_{fair}, T^{**}_{fair}, and T^*_{wfair}, T^{**}_{wfair}. These transformations remain correct under the process semantics. For example, we have

$$\Pi_{\text{fair}}[\![S, \text{Var}(S)]\!] = \Pi[\![T^*_{\text{fair}}(S), \text{Var}(S)]\!]$$

for every $L(\|)$-program S.

Correctness Under Fairness Assumptions. Classical proof methods for parallel programs with shared variables like [24] or [19] deal with parallelism by arbitrary interleaving. At present, the main proof methods for fair parallelism are that of [25] and [22]. In both cases, fairness is studied in the context of temporal logic [29] which is able to express a richer class of program properties than the input-output properties considered in Section 6.

A connection between fairness and temporal logic may seem natural because fairness can be expressed in temporal logic. For example, with □ denoting "always" and ◇, "eventually," □◇ expresses "infinitely often," and hence

$$\Box\Diamond(i \text{ is enabled}) \rightarrow \Box\Diamond(i \text{ is activated})$$

expresses strong fairness for component i [29]. Indeed, the approach of [25] is to express fairness in terms of such formulas and use them directly in the correctness proofs of parallel programs.

On the other hand, the approach of [23] shows that dealing with fairness is quite independent of temporal logic. Though in [23] temporal logic is used to express the desired program properties, proofs of these properties use well-founded arguments that are especially tailored to the different fairness assumptions.

The emphasis in our paper was on program transformations that reduce fair parallelism to ordinary parallelism. Since these transformations are correct both under the "termination" semantics $\mathscr{M}$ and the process semantics Π, they may be combined with *any* existing proof method for parallelism. We demonstrated this in Section 6 for the classical Owicki–Gries method [24]. However, since our transformations preserve the parallel structure of the original program (Definition 4.2) they may also be useful in combination with the compositional proof systems for parallel programs that are currently under development (e.g., [6, 30]).

Compositional means truly syntax-directed, so that, unlike in [19], [23], [24], and [25], no global test of interference freedom as in [24] is needed.

Implementation. At the University of Kiel, an interactive system has been implemented which generates the computations of parallel programs with shared variables [32]. The user can choose whether the current configuration should be displayed after each transition step in the computation or only at certain breakpoints set in the program text. For the interpretation of parallelism the user may select among the schedulers discussed in this paper: FAIR, RORO, and QUEUE. The system runs on an Apollo/Domain workstation, it is written in Standard Pascal augmented with machine-dependent calls of the Apollo window system, and it is used for teaching purposes.

Communicating Processes. We studied fairness only in the context of parallel programs with shared variables. However, our results on fair schedulers are independent of this particular syntax. In [4] these results are used in investigations of fairness for distributed, communicating processes. More on fairness in communicating processes can be found in [15].

ACKNOWLEDGMENT

The authors wish to thank A. Pnueli and the three referees for their helpful comments. W. Hesselink drew our attention to the note [10].

REFERENCES

1. APT, K. R., AND OLDEROG, E.-R. Proof rules and transformations dealing with fairness. *Sci. Comput. Program. 3* (1983), 65–100.
2. APT, K. R., AND OLDEROG, E.-R. Transformations realizing fairness assumptions in parallel programs (invited paper). In *Proceedings of the 1st Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 166*, M. Fontet and K. Mehlhorn, Eds., Springer-Verlag, Berlin, 1984, 26–42.
3. APT, K. R., AND PLOTKIN, G. D. Countable nondeterminism and random assignment. *J. ACM 33*, 4 (Oct. 1986), 724–767.
4. APT, K. R., FRANCEZ, N., AND KATZ, S. Appraising fairness in languages for distributed programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages* (Munich, Jan. 21–23, 1987). ACM, New York, 1987, 189–198.
5. APT, K. R., PNUELI, A., AND STAVI, J. Fair termination revisited—with delay. *Theor. Comput. Sci. 33* (1984), 65–84.
6. BARRINGER, H., KUIPER, R., AND PNUELI, A. A really abstract concurrent model and its temporal logic. In *Proceedings of ACM Symposium on Principles of Programming Languages* (St. Petersburg Beach, Fla., Jan. 13–15, 1986). ACM, New York, 1986, 173–183.
7. BOOM, H. J. A weaker precondition for loops. *ACM Trans. Program. Lang. Syst. 4*, 4 (Oct. 1982), 668–677.
8. BROY, M. Transformational semantics for concurrent programs. *Inf. Process. Lett. 11* (1980), 87–91.
9. BROY, M. Are fairness assumptions fair? In *Proceedings of the 2nd International Conference on Distributed Computing Systems* (Paris, 1981). IEEE, New York, 1981, 116–125.
10. DIJKSTRA, E. W. A class of allocation strategies inducing bounded delays only. Manuscript, *EWD 319* (1971).
11. DIJKSTRA, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
12. FISCHER, M. J., AND PATERSON, M. S. Storage requirements for fair scheduling. *Inf. Process. Lett. 17* (1983), 249–250.

13. FLON, L., AND SUZUKI, N. Nondeterminism and the correctness of parallel programs. In *Formal Description of Programming Concepts I*, E. J. Neuhold, Ed., North-Holland, Amsterdam, 1978, 598–608.
14. FLON, L., AND SUZUKI, N. The total correctness of parallel programs. *SIAM J. Comput. 10* (1981), 227–246.
15. FRANCEZ, N. *Fairness*. Springer-Verlag, Berlin, 1986.
16. GRIES, D. *The Science of Programming*. Springer-Verlag, New York, 1981.
17. HENNESSY, M. C. B., AND PLOTKIN, G. D. Full abstraction for a simple programming language. In *Proceedings of the 8th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 74*, J. Bečvář, Ed., Springer-Verlag, Berlin, 1979, 108–120.
18. LAMPORT, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM 17*, 8 (Aug. 1974), 453–455.
19. LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. 3* (1977), 125–143.
20. LAMPORT, L. What good is temporal logic? In *IFIP Information Processing 83*, R. E. A. Mason, Ed., North-Holland, Amsterdam, 1983, 657–668.
21. LEHMANN, D., PNUELI, A., AND STAVI, J. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 115*, S. Even and O. Kariv, Eds., Springer-Verlag, Berlin, 1981, 264–277.
22. MANNA, Z., AND PNUELI, A. Verification of concurrent programs: The temporal framework. In *The Correctness Problem in Computer Science, International Lecture Series in Computer Science*, R. S. Boyer and J. S. Moore, Eds., Academic Press, London, 1981.
23. MANNA, Z., AND PNUELI, A. Adequate proof principles for invariance and liveness properties of concurrent programs. *Sci. Comput. Program. 4* (1984), 257–289.
24. OWICKI, S., AND GRIES, D. An axiomatic proof technique for parallel programs. *Acta Inf. 6* (1976), 319–340.
25. OWICKI, S., AND LAMPORT, L. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst. 4*, 4 (July 1982), 455–495.
26. PARK, D. On the semantics of fair parallelism. In *Proceedings of Abstract Software Specifications, Lecture Notes in Computer Science 86*, D. D. Bjørner, Ed., Springer-Verlag, Berlin, 1979, 504–526.
27. PARK, D. A predicate transformer for weak fair iteration. In *Proceedings of the 6th IBM Symposium on Mathematical Foundations in Computer Science* (Hakone, Japan, 1981). IBM, 1981, 257–275.
28. PLOTKIN, G. D. A structural approach to operational semantics. Tech. Rep. DAIMI-FN 19, *Computer Science Dept.*, Aarhus Univ., Aarhus, Denmark, 1981.
29. PNUELI, A. The temporal semantics of concurrent programs. *Theor. Comput. Sci. 13* (1981), 45–60.
30. STIRLING, C. A compositional reformulation of Owicki–Gries' partial correctness logic for a parallel while language. In *The 13th Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 226*, L. Kott, Ed., Springer-Verlag, Berlin, 1986, 407–415.
31. TSICHRITZIS, D. C., AND BERNSTEIN, P. A. *Operating Systems*. Academic Press, Orlando, Fla., 1974.
32. WOLFF, T. Implementation of a transition semantics for parallel programs with shared variables. Diploma thesis (in German). Institut für Informatik, Univ. Kiel, Kiel, West Germany, 1987.

Received July 1984; revised July 1986 and December 1987; accepted December 1987