TRANSFORMATIONS REALIZING FAIRNESS ASSUMPTIONS FOR PARALLEL PROGRAMS

K.R. Apt
LITP
Université Paris VII
75251 Paris

E.-R. Olderog
Institut für Informatik
Universität Kiel
D-2300 Kiel 1

Abstract. Parallel programs with shared variables are studied under a semantics which assumes the fair execution of all parallel components. We present transformations which reduce this fair semantics to a simple interleaving semantics with help of random assignments z:=? . In fact, different notions of fairness are considered: impartiality, liveness, weak and strong fairness. All transformations preserve the structure of the original programs and are thus suitable as a basis for syntax-directed correctness proofs.

1. Introduction

This paper considers parallel programs $S = S_1 \| \ldots \| S_n$ where the components S_i of S are sequential programs which communicate with each other implicitly via shared variables. The correctness properties and (in-)formal reasoning about such programs depend on the semantical notion of execution of S.

The simplest way of modelling the execution of S is by arbitrary interleaving of the execution sequences of its components S_i /Br1,Br2,FS1,FS2/. But in general interleaving is not what we wish to express when writing $S = S_1 \| \ldots \| S_n$ since it models only the concept of multiprogramming where S runs on a single processor /MP/.

Here we investigate the more ambitious idea of a truly concurrent execution of S where every component S_i runs on its own processor. To formalize this idea we follow the proposal of /MP,OL/ to model concurrency of S by interleaving the execution sequences of its components, but with the additional assumption of fairness. Informally, fairness states that every component S_i of S which is sufficiently often enabled will eventually progress. Different interpretations of "sufficiently often enabled" give rise to different notions of fairness, viz. impartiality /LPS/, liveness /OL/, weak and strong fairness /AO/. For liveness e.g. "sufficiently often enabled" is interpreted as "not yet terminated".

So far semantics and proof theory for fairness assumptions have been studied mainly in the context of nondeterministic do-od-programs (see /Fr/ for an overview). For parallel programs $S = S_1 \| \ldots \| S_n$ the question of fairness has been dealt with only by translating the given program S back into a nondeterministic do-od-program /APS,LPS/ or by resorting to methods of temporal logic /OL/ which often requires a translation of the original program S into an equivalent formula in temporal logic /MP,Pn/.

In our paper we present a series of transformations T which reduce the concurrent or fair semantics of parallel programs to the simple interleaving semantics with help of random assignments z:=? /AP/, one transformation for each notion of fairness. All transformations preserve the parallel structure of the original programs. The approach represents a refinement of the transformation technique introduced in /AO/ for nondeterministic do-od-programs.

The interest in such transformations T is twofold:

(1) T can be considered as a sort of scheduler which guarantees that the resulting program T(S) realizes exactly all the fair executions of S.

(2) T can be used as a basis for syntax-directed correctness proofs. The idea is to apply an extension of the proof system of /OG/ dealing also with random assignments to T(S).

In this paper we concentrate on the first aspect. We state a number of results on the existence or non-existence of transformations with paricular properties. We hope that these results give a better insight into the structure of the various notions of fairness. Proofs will appear in the full version of this paper.

2. Parallel Programs

We assume sets Var of variables ranging over integers, Exp of expressions and Bex of Boolean expressions with typical elements $x,y,z \in Var$, $s,t \in Exp$ and $b,c \in Bex$. Sequential programs are defined by the following BNF-like syntax:

$$S ::= skip \mid x:=t \mid z:=? \mid S_1;S_2 \mid \underline{if}\ b\ \underline{then}\ S_1\ \underline{else}\ S_2\ \underline{fi} \mid$$
$$\underline{while}\ b\ \underline{do}\ S_1\ \underline{od} \mid \underline{await}\ b\ \underline{then}\ S_1\ \underline{end}$$

where for simplicity nested while's and await's are disallowed. Let $\underline{if}\ b\ \underline{then}\ S_1\ \underline{fi}$ abbreviate $\underline{if}\ b\ \underline{then}\ S_1\ \underline{else}$ skip $\underline{fi}$.

Besides usual assignments of the form x:=t we consider random assignments z:=? which assign an arbitrary non-negative integer to z /AP/. Thus z:=? is an explicit form of unbounded nondeterminism in the sense that termination of z:=? is guaranteed but infinitely many final states are possible /Pa /.

Await-statements $S = \underline{await}\ b\ \underline{then}\ S_1\ \underline{end}$ are used to achieve synchronization in the context of parallel composition. S is executed only if b is true. What makes it different from $\underline{if}\ b\ \underline{then}\ S_1\ \underline{fi}$ is that the await guarantees that S_1 is executed as an indivisible action /OG/ (cf. Sec.3).

By a parallel program we mean a program of the form

$$S = S_0;(S_1 \parallel \ldots \parallel S_n)$$

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 (parallel) components of S inside the parallel composition $S_1 \parallel \ldots \parallel S_n$.

We distinguish four classes of programs: L(‖), L(‖,?), L(‖,await) and L(‖,await,?) depending on whether random assignments or/and await-statements are used. L(‖,await) is essentially the language studied in /OG/.

In our paper we will study certain (program) transformations, i.e. mappings

$$T: L(\|) \ [\text{or } L(\|,\text{await})] \longrightarrow L(\|,\text{await},?).$$

Sometimes it is more convenient to leave certain details of such a transformation open. To this end, we consider transformation schemes, i.e. mappings

$$\mathbb{T}: L(\|) \ [\text{or } L(\|,\text{await})] \longrightarrow \mathcal{P}(L(\|,\text{await},?)) \setminus \{\emptyset\}$$

which assign to every program S a non-empty set of transformed programs $S' \in \mathbb{T}(S)$. ($\mathcal{P}(M)$ is the power set of a set M.) By selecting a particular $S' \in \mathbb{T}(S)$ for every program S we obtain a so-called instance T of $\mathbb{T}$. This is a transformation T as above.

3. Interleaving Semantics

We take an interpretation with integers as domain $\mathcal{D}$ assigning the standard meaning to all symbols of Peano arithmetic. The set of (proper) states is given by $\Sigma = Var \longrightarrow \mathcal{D}$ with typical elements σ, τ. Notations like $\sigma[d/x]$, $\sigma\restriction X$ and $\sigma(b)$ are as usual. We add 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 $S \in L(\|,\text{await},?)$ and a state σ. Following /HP,Pl/ we introduce a transition relation $\longrightarrow$ between configurations. $\langle S,\sigma\rangle \longrightarrow \langle S_1,\sigma_1\rangle$ 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 the empty program E with E;S = S;E = E.

The relation $\longrightarrow$ is defined by structural induction on L(‖,await,?). Typical clauses are:

a) $\langle \text{skip},\sigma\rangle \longrightarrow \langle E,\sigma\rangle$

b) $\langle z:=?,\sigma\rangle \longrightarrow \langle E,\sigma[d/x]\rangle$ for every $0 \leq d \in \mathcal{D}$.

c) $\langle \underline{\text{while}}\ b\ \underline{\text{do}}\ S_1\ \underline{\text{od}},\sigma\rangle \longrightarrow \langle S_1;\underline{\text{while}}\ b\ \underline{\text{do}}\ S_1\ \underline{\text{od}},\sigma\rangle$ if $\sigma(b) = \text{true}$.

d) $\langle \underline{\text{while}}\ b\ \underline{\text{do}}\ S_1\ \underline{\text{od}},\sigma\rangle \longrightarrow \langle E,\sigma\rangle$ if $\sigma(b) = \text{false}$.

e) $\langle \underline{\text{await}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{end}},\sigma\rangle \longrightarrow \langle E,\tau\rangle$ if $\sigma(b) = \text{true}$ and $\langle S_1,\sigma\rangle \longrightarrow^* \langle E,\tau\rangle$ where $\longrightarrow^*$ denotes the reflexive, transitive closure of $\longrightarrow$.

f) If $\langle S_1,\sigma\rangle \longrightarrow \langle S_2,\tau\rangle$ then $\langle S_1;S,\sigma\rangle \longrightarrow \langle S_2;S,\tau\rangle$

g) If $\langle S_i,\sigma\rangle \longrightarrow \langle T_i,\tau\rangle$ then
$\langle S_1 \| \ldots \| S_n,\sigma\rangle \longrightarrow \langle S_1 \| \ldots \| S_{i-1} \| T_i \| S_{i+1} \| \ldots \| S_n,\tau\rangle$.

Note that assignments, evaluations of Boolean expressions, and await-statements are executed 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 $\longrightarrow$ we introduce some further concepts. A configuration $\langle S,\sigma\rangle$ is maximal if it has no successor w.r.t. $\longrightarrow$. A terminal configuration is a maximal configuration $\langle S,\sigma\rangle$ with $S = E \| \ldots \| E$. All other maximal configurations are called deadlocked. A computation of S (starting in σ) is a finite or infinite sequence

$$\xi : \langle S,\sigma\rangle \longrightarrow \langle S_1,\sigma_1\rangle \longrightarrow \ldots \longrightarrow \langle S_k,\sigma_k\rangle \longrightarrow \ldots$$

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

$$\xi : \langle S,\sigma\rangle \longrightarrow \ldots \longrightarrow \langle T,\tau\rangle$$

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 interleaving semantics of programs $S \in L(\|, \text{await}, ?)$ is

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

defined by

$$\begin{aligned}\mathcal{M}[\![S]\!](\sigma) = \quad & \{\tau \mid \langle S,\sigma\rangle \longrightarrow^* \langle E \| \ldots \| E,\tau\rangle\} \\ & \cup \{\bot \mid S \text{ can diverge from } \sigma\} \\ & \cup \{\Delta \mid S \text{ can deadlock from } \sigma\}\end{aligned}$$

We also consider a variant of $\mathcal{M}$ ignoring deadlocks:

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

Some further notions. The component S_i has terminated in $\langle S_1 \| \ldots \| S_n,\sigma\rangle$ if $S_i = E$. The component S_i is disabled in $\langle S_1 \| \ldots \| S_n,\sigma\rangle$ if either $S_i = E$ or $S_i =$ await b then S end; T with $\sigma(b) =$ false. The component S_i is enabled if it is not disabled, i.e. if S_i is not terminated and whenever $S_i =$ await b then S end;T holds then $\sigma(b) =$ true. The component S_i is active in the step $\langle S_1 \| \ldots \| S_n,\sigma\rangle \longrightarrow \langle T_1 \| \ldots \| T_n,\tau\rangle$ if $\langle S_i,\sigma\rangle \longrightarrow \langle T_i,\tau\rangle$. A program S is deadlock-free if $\mathcal{M}[\![S]\!] = \mathcal{M}_{-\Delta}[\![S]\!]$.

4. Impartiality

Consider the program

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

Under the interleaving semantics S can diverge: $\bot \in \mathcal{M}[\![S^*]\!](\sigma)$ if $\sigma(b) =$ true. However, in every concurrent or "fair" computation of S^* the second component S_2 will eventually be executed causing termination of S_1 and hence S^* itself. The question is how to capture this intuitive notion of fairness.

In this section we define a first approximation to the concept of fairness, viz. impartiality /LPS/.

Definition 4.1 A computation $\xi: \langle S,\sigma\rangle = \langle T_1,\sigma_1\rangle \to \ldots \to \langle T_j,\sigma_j\rangle \to \ldots$ of an $L(\|,\text{await})$-program $S = S_0;(S_1 \| \ldots \| S_n)$ is impartial if ξ is finite or for every $i \in \{1,\ldots,n\}$ there are infinitely many j such that component S_i is active in step $\langle T_j,\sigma_j\rangle \to \langle T_{j+1},\sigma_{j+1}\rangle$.

Thus in an infinite impartial computation every component will eventually progress. The concurrent semantics of programs $S \in L(\|,\text{await})$, modelled here by interleaving and the assumption of impartiality, is now given by

$$\mathcal{M}_{imp}[\![S]\!](\sigma) = \{\tau \mid \langle S,\sigma\rangle \to^* \langle E \| \ldots \| E,\tau\rangle\} \cup \left\{\bot \;\middle|\; \begin{array}{l}\exists \text{ infinite impartial computation of } S\\ \text{starting in } \sigma\end{array}\right\} \cup \{\Delta \mid S \text{ can deadlock from } \sigma\}$$

To see the impact of this definition, let us look at the example S^* again. Under the assumption of impartiality S^* always terminates: $\bot \notin \mathcal{M}_{imp}[\![S^*]\!](\sigma)$ for every state σ.

5. Structure Preserving Transformations

In this section we restrict ourselves to programs in $L(\|)$. Our aim is to find a transformation T which reduces the concurrent semantics $\mathcal{M}_{imp}$ of $L(\|)$ to the ordinary interleaving semantics $\mathcal{M}$, i.e. with

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

Such transformations T are useful for two reasons: firstly, they describe a class of schedulers which implement true concurrency on a single processor machine, and secondly, they provide a systematic approach of refining existing proof methods for program correctness under interleaving semantics to methods for dealing directly with concurrent semantics. Of course, we cannot expect the transformed programs T(S) to be in $L(\|)$ because $\mathcal{M}_{imp}$ introduces unbounded nondeterminism (and thus discontinuous semantic operators /Di/) as opposed to $\mathcal{M}$. But we can control this unbounded nondeterminism by making it explicit via random assignments

z:=?

as analysed in /AP/.

First attempt

A simple way of reducing concurrency to interleaving is to combine two types of already existing transformations. Given a parallel program $S = S_0;(S_1 \| \ldots \| S_n)$ one first follows the approach of /FS1,FS2/ or /Br1,Br2/ and translates S into a big nondeterministic do-od-program $T_{nd}(S)$ which makes the interleaving semantics $\mathcal{M}[\![S]\!]$ syntactically visible. Then one can apply the transformations T_{fair} of /AO/ to $T_{nd}(S)$ which use random assignments to realize the assumption of fairness in the

context of do-od-programs. The drawback of this solution is that the first translation T_{nd} destroys the syntactic structure of programs S.

Instead we are interested in transformations which preserve the parallel struture of programs.

Definition 5.1 A transformation $T:L(\|)\to L(\|,\underline{await},?)$ is called $\|$-preserving if T satifies

$$T(\ S_o;(S_1\|\ldots\|S_n)\ ) = T_o^n(S_o);(\ T_1^n(S_1)\ \|\ \ldots\ \|\ T_n^n(S_n)\ )$$

where T_i^n is a sub-transformation working on the i-th component of S. The notation implies 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. A transformation scheme $\mathbb{T}$ is $\|$-preserving if every instance T of $\mathbb{T}$ is $\|$-preserving.

Second attempt

In /AO/ we showed that in the context of Dijkstra's nondeterministic do-od-programs fairness assumptions can be realized by just adding random assignments z:=? and refining Boolean expressions in a certain "admissible" way. The question arises whether this is also possible for parallel programs $S\in L(\|)$.

Definition 5.2 A transformation $T:L(\|)\to L(\|,?)$ is admissible 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:=? and z:=t inside of S

(2) in Boolean conjuncts c used to strengthen Boolean expressions b of loops while b do S_1 od or conditionals if b then S_1 else S_2 fi in S. We require that this stregthening is done schematically, i.e. the conjunct c in independent of the actual form of b.

Again a transformation scheme $\mathbb{T}$ is admissible if every instance T of $\mathbb{T}$ is.

Note that because T(S) manipulates additional variables Z the best we can hope to prove is that $\mathcal{M}_{imp}[\![S]\!]$ agrees with $\mathcal{M}[\![T(S)]\!]$ "modulo Z". This notion is defined as follows: for states $\sigma\in\Sigma$ and sets $Z\subseteq Var$ of variables let

$$\sigma\setminus Z = \sigma\restriction(Var\setminus Z).$$

This notation is extended to sets $M\subseteq\Sigma\cup\{\bot,\Delta\}$ pointwise:

$$M\setminus Z = \{\sigma\setminus Z\mid\sigma\in M\}\cup\{\bot\mid\bot\in M\}\cup\{\Delta\mid\Delta\in M\}.$$

For state transformers $\mathcal{M}_1,\mathcal{M}_2:\Sigma\to\mathcal{P}(\Sigma\cup\{\bot,\Delta\})$ we write

$$\mathcal{M}_1 = \mathcal{M}_2 \ \underline{\text{mod}}\ Z$$

if $\mathcal{M}_1(\sigma)\setminus Z = \mathcal{M}_2(\sigma)\setminus Z$ holds for every state $\sigma \in \Sigma$.

Theorem 5.3 There is no admissible transformation $T:L(\|)\rightarrow L(\|,?)$ such that for every program $S\in L(\|)$

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

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

The theorem states that it is more difficult to find transformations T realizing fairness (here impartiality) for parallel programs than for nondeterministic ones. The reason is that transformations $T:L(\|)\rightarrow L(\|,?)$ would have to terminate the presently executed component S_i of a program $S\in L(\|)$ in order to force a shift of control to another component S_j. But after terminating S_i there is no possibility of resuming S_i later on. To achieve this effect we necessarily need an additional language construct in T: the await-statement.

Third attempt

First we extend Definition 5.2 of admissibility to transformations $T:L(\|)\rightarrow L(\|,await,?)$ also

(3) new await-statements await c then S_1 end where S_1 is a sequence of assignments of the form (1) of Definition 5.2.

To conduct a finer analysis, we introduce further concepts:

Definition 5.4 A transformation $T:L(\|)\rightarrow L(\|,await,?)$ is sequential (in every component) if it is admissible, i.e. if it is of the form

$$T(\, S_o;(S_1 \| \ldots \| S_n)\,) = T_o^n(S_o);(\, T_1^n(S_1) \| \ldots \| T_n^n(S_n)\,) ,$$

and if it preserves sequential composition in every component, i.e. if for every $i = 1,\ldots,n$

$$T_i^n(S_1' ; S_2') = T_i^n(S_1') ; T_i^n(S_2')$$

holds. A transformation scheme $\mathbb{T}$ is sequential if every instance T of $\mathbb{T}$ is.

Sequentiality yields particularly simple transformations.

Definition 5.5 A transformation $T:L(\|)\rightarrow L(\|,await,?)$ is faithful if T does not introduce deadlocks, i.e. $\Delta \notin \mathcal{M}[\![T(S)]\!](\sigma)$ holds for every $S\in L(\|)$ and $\sigma\in\Sigma$. A transformation scheme $\mathbb{T}$ is faithful if every instance T of $\mathbb{T}$ is. Otherwise T and $\mathbb{T}$ are called deadlocking.

Transformations implementing schedulers should be faithful as schedulers should never run into any deadlocked configuration. The notion of a faithful transformation was first introduced for nondeterministic do-od-programs in /AO/ where it meant

absence of guard-failures. Unfortunately, we cannot find a simple sequential transformation which is faithful:

Theorem 5.6 There is no faithful, sequential transformation $T: L(\|) \to L(\|, \underline{await}, ?)$ such that for every program $S \in L(\|)$

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

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

Faithful, but non-sequential transformations will be presented in Sec. 6.

A Solution

However, we can find a deadlocking transformation (scheme)

$$T_{imp+\Delta} : L(\|) \longrightarrow \mathcal{P}(L(\|, \underline{await}, ?)) \setminus \{\emptyset\}$$

which is sequential and realizes impartiality. Certainly, deadlocking transformations T are not suitable as implementations of fair schedulers, but - as first observed in /APS/ - may lead to simplified correctness proofs of transformed programs T(S). This is why we are also interested in deadlocking transformations.

For a given program $S = S_0; (S_1 \| \ldots \| S_n)$ in $L(\|)$ let $T_{imp+\Delta}(S)$ be the set of all programs resulting from S by

(1) prefixing S with an initialisation part

$$INIT = z_1 := ?; \ \ldots \ ; z_n := ?$$

(2) replacing in every loop $\underline{while}$ b $\underline{do}$ S' $\underline{od}$ of a component S_i $\underline{some}$ atomic statement A in S' by

$$TEST_i(A) = \underline{await}\ \bar{z} \geqslant 1\ \underline{then}\ z_i := ?;\ \underline{for}\ j \neq i\ \underline{do}\ z_j := z_j - 1\ \underline{od};\ A\ \underline{end}$$

(for i = 1,...,n).

Here we use new variables $z_1, \ldots, z_n$ not already present in S and the following abbreviations:

$$\bar{z} \geqslant 1 = z_1 \geqslant 1 \wedge \ldots \wedge z_n \geqslant 1$$

$$\underline{for}\ j \neq i\ \underline{do}\ z_j := z_j - 1\ \underline{od} =$$

$$z_1 := z_1 - 1; \ldots; z_{i-1} := z_{i-1} - 1; z_{i+1} := z_{i+1} - 1; \ldots; z_n := z_n - 1$$

To see its impact, we apply $T_{imp+\Delta}$ to the program

$$S^* = \underline{while}\ b\ \underline{do}\ x := x+1\ \underline{od} \ \| \ b := false$$

of Sec. 4. Here there is exactly one program $T \in T_{imp+\Delta}(S)$, viz.

$T =$ $z_1:=?;\ z_2:=?;$
(while b do await $z_1, z_2 \geqslant 1$ then
$z_1:=?;\ z_2:=z_2-1;$
$x:=x+1$
end
od
$\|$ b:=false)

T uses variables z_1, z_2 for scheduling purposes. The variable z_2 counts how many times we may enter the while-loop of the first component of T without switching control to the second component. (The variable z_1 is introduced for analogous purposes but not relevant for this particular program T without a while-loop in its second component.) Initially z_2 is set to an arbitrary non-negative integer. Each time the while-loop is entered z_2 is decremented by 1. This ensures that this loop cannot be executed arbitrarily long without falsifying $z_1, z_2 \geqslant 1$. Note that as soon as the Boolean expression $z_1, z_2 \geqslant 1$ of the await-statement is false, it remains so even after executing the second component b:=false. Hence T can deadlock from states σ with $\sigma(b) = \text{true}$.

Thus $T_{imp+\Delta}$ is a deadlocking transformation which transforms all diverging non-impartial computations of S into deadlocking computations of T. Indeed $T_{imp+\Delta}$ realizes the assumption of impartiality in the sense of:

Theorem 5.7 For every L($\|$)-program S and $T \in T_{imp+\Delta}(S)$ the equation

$$\mathcal{M}_{imp}[\![S]\!] = \mathcal{M}_{-\Delta}[\![T]\!] \text{ mod } Z$$

holds where Z is the set of auxiliary variables in T.

6. Liveness

Consider the L($\|$)-program

$$S^{**} = \text{while b do } x:=x+1 \text{ od } \| \text{ skip}\ .$$

Intuitively, S^{**} should diverge for states σ with $\sigma(b) = \text{true}$ - independently whether the interleaving or a concurrent, i.e. "fair" semantics is chosen. However, with our definition of impartiality S^{**} always terminates: $\perp \notin \mathcal{M}_{imp}[\![S^{**}]\!](\sigma)$ for every state σ.

Thus impartility is not adequate to capture the idea of fairness even for L($\|$). Therefore we introduce the refined concept of liveness which distinguishes between terminated and running components of parallel programs.

Definition 6.1 A computation $\xi : \langle S,\sigma\rangle = \langle T_1,\sigma_1\rangle \to \ldots \to \langle T_j,\sigma_j\rangle \to \ldots$ of a program $S = S_0;(S_1 \| \ldots \| S_n)$ in $L(\|,\underline{await})$ is live if ξ is finite or the following holds for every $i \in \{1,\ldots,n\}$: either component S_i has terminated in some $\langle T_j,\sigma_j\rangle$ or there are infinitely many j such that component S_i is active in step $\langle T_j,\sigma_j\rangle \to \langle T_{j+1},\sigma_{j+1}\rangle$.

Thus in a live computation every non-terminated component will eventually be active and make progress. Analogously to $\mathcal{M}_{imp}$ we define a semantics $\mathcal{M}_{live}$ which captures this assumption of liveness:

$$\begin{aligned}\mathcal{M}_{live}[\![S]\!](\sigma) = \; & \{\tau \mid \langle S,\sigma\rangle \to^* \langle E \| \ldots \| E,\tau\rangle\} \\ & \cup \{\bot \mid \exists \text{ infinite } \underline{\text{live}} \text{ computation of } S \text{ starting in } \sigma\} \\ & \cup \{\Delta \mid S \text{ can deadlock from } \sigma\}\end{aligned}$$

Let us first establish an interesting relation between $\mathcal{M}_{imp}$ and $\mathcal{M}_{live}$.

Definition 6.2 A program S in $L(\|)$ is called strong if whenever $\langle S,\sigma\rangle \to^* \langle T_1 \| \ldots \| T_n,\tau\rangle$ with $T_i = E$ holds for some $i \in \{1,\ldots,n\}$ then $T_1 \| \ldots \| T_n$ cannot diverge from τ.

Informally, S cannot diverge with one component terminated. This property is not decidable but it is often easy to check whether a given program is strong. E.g. program S^* of Sec. 4 is strong.

Proposition 6.3 For strong programs S in $L(\|)$ the equation $\mathcal{M}_{live}[\![S]\!] = \mathcal{M}_{imp}[\![S]\!]$ holds.

As done for impartiality we are looking for structure preserving transformations which realize the assumption of liveness. Clearly, for strong $L(\|)$-programs we can use $T_{imp+\Delta}$ due to Proposition 6.3. But in general things are more complicated:

Theorem 6.4 There is no sequential transformation $T: L(\|) \to L(\|,\underline{await},?)$ such that for every $S \in L(\|)$

$$\mathcal{M}_{live}[\![S]\!] = \mathcal{M}_{-\Delta}[\![T(S)]\!] \;\; \underline{\text{mod}}\; Z$$

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

The result is based on the fact that sequential transformations T cannot distinguish whether a certain substatement is the final statement in a component of a parallel program or whether it is followed by some other statement. To accomplish this distinction we use in our transformations further auxiliary variables end_i which record termination of the component programs.

We present a faithful (but non-sequential) transformation scheme

$$T_{live}: L(\|) \longrightarrow \mathcal{P}(L(\|,\underline{await},?)) \setminus \{\emptyset\}.$$

For a given program $S = S_0;(S_1 \| \ldots \| S_n)$ in $L(\|)$ let $T_{live}(S)$ be the set of all programs resulting from S by

(1) prefixing S with an initialisation part

$$INIT = z_1:=?; \ldots;z_n:=?; end_1:=false; \ldots ;end:=false$$

(2) replacing in every loop $\underline{\text{while}}$ b $\underline{\text{do}}$ S' $\underline{\text{od}}$ of a component S_i $\underline{\text{some}}$ atomic statement A in S' by

$$\begin{aligned} TEST_i(A) = \ & \underline{\text{await}} \ turn = i \vee \overline{z} \geqslant 1 \ \underline{\text{then}} \\ & z_i:=?; \ \underline{\text{for}} \ j \neq i \ \underline{\text{do}} \\ & \quad \underline{\text{if}} \ \neg end_j \ \underline{\text{then}} \ z_j:=z_j-1 \ \underline{\text{fi}} \\ & \underline{\text{od}}; \\ & A \\ & \underline{\text{end}} \end{aligned}$$

(for i = 1,...,n).

(3) suffixing every component S_i by

$$END_i = end_i:=true$$

(for i = 1,...,n).

Again the z_i's and end_i's are new variables not already present in S. As additional abbreviation we use

$$turn = \min\{ j \mid z_j = \min\{ z_k \mid end_k = false \} \} \ .$$

Due to (3) all components of transformed programs $T \in T_{live}(S)$ have terminated when all variables end_i are true. Thus the expression turn is properly defined whenever a test $TEST_i(A)$ is executed in T.

Theorem 6.5 For every program $S \in L(\|)$ and $T \in T_{live}(S)$ the equation

$$\mathcal{M}_{live}[\![S]\!] = \mathcal{M}[\![T]\!] \ \underline{\text{mod}} \ Z$$

holds where Z is the set of auxiliary variables z_i and end_i in T.

The proof of Theorem 6.5 shows that the transformation scheme T_{live} not only models the input-output behaviour of S but in fact provides a one-one correspondence between live computations of S and arbitrary computations of $T \in T_{live}(S)$. Therefore we can view T_{live} as an abstract specification of schedulers which guarantee liveness for parallel programs S. By Theorem 6.5 every deterministic scheduler can be implemented by replacing the random assignments z:=? in $T \in T_{live}(S)$ by deterministic assignments and by refining the Boolean conjuncts "$\overline{z} \geqslant 1$" in await-statements $TEST_i(A)$. (See /Pa/ for the notion of implementation in the context of specifica-

tions with unbounded nondeterminism.) Moreover Theorem 6.5 guarantees that all these implemented schedulers are deadlock-free and therefore never require any rescuing or backtracking from deadlocked configurations /Ho/.

7. Weak Fairness

In this section we extend our programming language to L($\|$,await). Though liveness is an adequate formalization of the concept of concurrency for the language L($\|$), it is not sufficient for L($\|$,await). Consider the program

$$S^{***} = \underline{\text{while}}\ b\ \underline{\text{do}}\ x:=x+1\ \underline{\text{od}}\ \|\ \underline{\text{await}}\ \neg b\ \underline{\text{then}}\ \text{skip}\ \underline{\text{end}}\ .$$

We expect computations of S^{***} starting in a state σ with $\sigma(b)$ = true to diverge - independently whether the interleaving or a concurrent semantics is chosen. But with the simple defintion of liveness S^{***} always terminates: $\perp \notin \mathcal{M}_{live}[\![S^{***}]\!](\sigma)$ for every state σ.

In the presence of await's we have to refine the idea of liveness by replacing the notion of termination by the notion of enabledness of components (cf. Sec.3). This leads to the following concept of weak fairness /AO,FP/ (called justice in /LPS/).

Definition 7.1 A computation $\xi : \langle S, \sigma\rangle = \langle T_1,\sigma_1\rangle \rightarrow \ldots \rightarrow \langle T_j,\sigma_j\rangle \rightarrow \ldots$ of an L($\|$,await)-program $S = S_o;(S_1 \| \ldots \| S_n)$ is weakly fair if ξ is finite or the following holds for every $i \in \{1,\ldots,n\}$: if for all but finitely many j the component S_i is enabled in $\langle T_j,\sigma_j\rangle$, then there are infinitely many j such that component S_i is active in step $\langle T_j,\sigma_j\rangle \rightarrow \langle T_{j+1},\sigma_{j+1}\rangle$.

Thus in a weakly fair computation every component which is from some moment on continuously enabled will eventually make progress. This definition induces a semantics $\mathcal{M}_{wfair}$ analogously to $\mathcal{M}_{live}$.

Remark 7.2 $\mathcal{M}_{wfair}[\![S]\!] = \mathcal{M}_{live}[\![S]\!]$ holds for all programs $S \in L(\|)$.

As for impartiality and liveness we wish to develop transformations which realize the assumption of weak fairness. (Note that the previous definitions of $\|$ -preserving, admissible, sequential, faithful and deadlocking have straightforward extensions to transformations $T : L(\|,\underline{\text{await}}) \rightarrow L(\|,\underline{\text{await}},?)$.) These transformations are again more sophisticated than the previous ones because we have to check enabledness of components in front of every atomic statement inside of while-loops.

We refine the transformation scheme T_{live} of Sec.6 to an admissible, faithful scheme T_{wfair}. (Clearly T_{wfair} cannot be sequential by Theorem 6.4 and Remark 7.2.) Given a program $S = S_o;(S_1 \| \ldots \| S_n)$ in L($\|$,await) this scheme T_{wfair} will use sets of new variables, viz. z_i, end_i, pc_i for i = 1,...,n. The z_i's and end_i's are used as in T_{live}. The pc_i's are a restricted form of program counters which indicate whe 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 $l \geqslant 1$ 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 labelled by l. Further on we introduce for S_i the abbreviation

$$\text{enabled}_i = \neg \text{end}_i \wedge \bigwedge_{l \in L_i} (pc_i = l \longrightarrow b_l) \ .$$

By the following construction of T_{wfair}, enabled_i will be true iff the component S_i of S is indeed enabled.

The transformation scheme

$$T_{wfair} : L(\|, \underline{await}) \longrightarrow \mathcal{P}(L(\|, \underline{await}, ?)) \setminus \{\emptyset\}$$

maps a given program $S = S_0; (S_1 \| \ldots \| S_n)$ in $L(\|)$ into the set $T_{wfair}(S)$ of all programs resulting from S by

(1) prefixing S with

$$\text{INIT} = \underline{\text{for}}\ i = 1,\ldots,n\ \underline{\text{do}}\ z_i := ?;\ \text{end}_i := \text{false};\ pc_i := 0\ \underline{\text{od}}$$

(2) replacing every substatement $\underline{\text{await}}\ b_l\ \underline{\text{then}}\ S'\ \underline{\text{end}}$ with $l \in L_i$ in S_i by

$$pc_i := l;\ \underline{\text{await}}\ b_l\ \underline{\text{then}}\ S';\ pc_i := k\ \underline{\text{end}}$$

where $k \notin L_i$ holds, e.g. $k = 0$ (for $i = 1,\ldots,n$).

(3) transforming in the so prepared program every loop $\underline{\text{while}}\ b\ \underline{\text{do}}\ S'\ \underline{\text{od}}$ in a component S_i as follows:

(i) replace some atomic statement A in S' by

$$\begin{aligned}
\text{TEST}_i(A) = \ & \underline{\text{await}}\ \text{turn} = i \vee \overline{z} \geqslant 1\ \underline{\text{then}} \\
& z_i := ?;\ \underline{\text{for}}\ j \neq i\ \underline{\text{do}} \\
& \quad \underline{\text{if}}\ \text{enabled}_j\ \underline{\text{then}}\ z_j := z_j - 1 \\
& \quad \underline{\text{else}}\ z_j := ?\ \ \underline{\text{fi}} \\
& \underline{\text{od}}; \\
& A \\
& \underline{\text{end}}
\end{aligned}$$

(ii) replace every other atomic statement B in S' not affected under (i) by

$$\begin{aligned}
\text{RESET}_i(B) = \ & \underline{\text{await}}\ \text{true}\ \underline{\text{then}} \\
& \underline{\text{for}}\ j \neq i\ \underline{\text{do}}\ \underline{\text{if}}\ \neg \text{enabled}_j\ \underline{\text{then}}\ z_j := ?\ \underline{\text{fi}}\ \underline{\text{od}}; \\
& B \\
& \underline{\text{end}}
\end{aligned}$$

Comment: If A or B are already await-statements, we "amalgamate" their Boolean expressions with turn = $i \wedge \overline{z} \geqslant 1$ or true to avoid nested await's. The expression turn is here defined as follows:

$$\text{turn} = \min\{j \mid z_j = \min\{z_k \mid \text{enabled}_k = \text{true}\}\}.$$

(4) suffixing every component S_i by

$$\text{END}_i = \text{end}_i = \text{true}$$

Inside $\text{RESET}_i(B)$ and $\text{TEST}_i(A)$ each of the variables z_j associated with S_j is reset as soon as S_j gets disabled. Thus z_j is continuously decremented by $z_j := z_j - 1$ inside $\text{TEST}_i(A)$ only if S_j is continuously enabled. This formalizes the idea of weak fairness where only those components which are continuously enabled are guaranteed to progress eventually.

Theorem 7.3 For every program $S \in L(\|, \underline{\text{await}})$ and $T \in T_{\text{wfair}}(S)$ the equation

$$\mathcal{M}_{\text{wfair}}[\![S]\!] = \mathcal{M}[\![T]\!] \ \underline{\text{mod}}\ Z$$

holds where Z is the set of auxiliary variables z_i, end_i and pc_i in T.

8. Strong Fairness

Weak fairness guarantees progress of those components which are continuously enabled. A more ambitious version of fairness is strong fairness /AO,FP/ (called fairness in /LPS/) where progress is already guaranteed if the component is infinitely often enabled.

Definition 8.1 A computation $\xi : \langle S,\sigma\rangle = \langle T_1,\sigma_1\rangle \to \ldots \to \langle T_j,\sigma_j\rangle \to \ldots$ of an $L(\|, \underline{\text{await}})$-program $S = S_0;(S_1 \| \ldots \| S_n)$ is strongly fair if ξ is finite or the following holds for every $i \in \{1,\ldots,n\}$: if for infinitely many j the component S_i is enabled in $\langle T_j,\sigma_j\rangle$, then there are infinitely many j such that component S_i is active in the step $\langle T_j,\sigma_j\rangle \to \langle T_{j+1},\sigma_{j+1}\rangle$.

Analogously to $\mathcal{M}_{\text{wfair}}$ we define $\mathcal{M}_{\text{sfair}}$. We refine the previous transformation scheme T_{wfair} to a scheme

$$T_{\text{sfair}} : L(\|, \underline{\text{await}}) \longrightarrow \mathcal{P}(L(\|, \underline{\text{await}}, ?)) \setminus \{\emptyset\}$$

for strong fairness. For a given program $S = S_0;(S_1 \| \ldots \| S_n)$ in $L(\|, \underline{\text{await}})$ let $T_{\text{sfair}}(S)$ result from S by applying the steps (1), (2) and (4) as in T_{wfair} but with the following new step (3):

(3) replace in the so prepared program every atomic statement A occurring in a while-loop of S_i by

$TEST_i(A)$ = await turn = i $\vee$ $\bar{z} \geqslant 1$ then

z_i:=?; for j $\neq$ i do

if enabled$_j$ then z_j:=z_j-1 fi

od;

A

end

As with T_{wfair} we have to test the enabledness of the components S_j of S in front of every atomic statement. But in contrast to T_{wfair} the transformed programs $T \in T_{sfair}(S)$ do not reset the variables z_j for component S_j when S_j gets disabled. Instead we have to decrement z_j (and be prepared for switching to component S_j) whenever S_j is enabled. This change ensures that those S_j which are infinitely often enabled make eventually progress. These observations are formalized in:

Theorem 8.2 For every program $S \in L(\|, \text{await})$ and $T \in T_{sfair}(S)$ the equation

$$\mathcal{M}_{sfair}[\![S]\!] = \mathcal{M}[\![T]\!] \text{ mod } Z$$

holds where Z is the set of auxiliary variables z_i, end_i and pc_i in T.

9. Conclusion

We presented here a series of structure preserving transformations which reduce different notions of a concurrent or fair semantics for parallel programs to a simple interleaving semantics. These transformations can be viewed as abstract specifications of schedulers guaranteeing only fair computations.

But they also provide a basis for syntax-directed correctness proofs for parallel programs under fairness assumptions. We outline this idea with help of a simple example. Consider the L($\|$)-program

$$S = \text{while } x > 0 \text{ do skip; } x:=x-1 \text{ od} \;\|\; \text{while } x > 0 \text{ do skip od}\ .$$

Under the interleaving semantics $\mathcal{M}$ this program can diverge but under the semantics $\mathcal{M}_{live}$ modelling the assumption of liveness S always terminates. We write this fact as

(1) $$\models_{live} \{\text{true}\}\ S\ \{\text{true}\}$$

in the sense of total correctness modulo liveness.

First observe that S is a strong L($\|$)-program. Thus to prove (1) it suffices to apply the transformation scheme $T_{imp+\Delta}$ modelling impartiality and prove for some program $T \in T_{imp+\Delta}(S)$

(2) $$\models_{-\Delta} \{\text{true}\}\ T\ \{\text{true}\}$$

where $\models_{-\Delta}$ refers to the interleaving semantics $\mathcal{M}_{-\Delta}$ ignoring deadlocks. The equivalence of (1) and (2) follows from Theorem 5.7 and Proposition 6.3.

To prove (2) we will use a simple extension of the proof system /OG/ which ignores deadlocks but deals with termination in the presence of random assignments z:=? , i.e. the extension deals with "total correctness modulo deadlocks". As in /OG/ the extended proof system proceeds in two steps: first it proves correctness of the components of a parallel program T and then it uses a proof rule for parallel composition to prove correctness of the whole program T.

Note that in our particular example S there are two transformed programs $T \in T_{imp+\Delta}(S)$ for which we could prove correctness in the sense of (2) with the extended proof system of /OG/ - one, say T_1, is obtained by applying the expansion $TEST_1(A)$ of $T_{imp+\Delta}(S)$ to the atomic statement A = skip in the first component of S, another one, say T_2, by applying $TEST_1(A)$ to A = x:=x-1. It turns out that for T_2 claim (2) is considerably simpler to prove correct in the extended proof system than for T_1. This observation explains the advantage of having nondeterministic transformation <u>schemes</u> like $T_{imp+\Delta}$ to our disposal: they can be applied flexible according to the needs of particular examples like S.

Finally, we stress the fact that for proving (1) about S we simply need to prove total correctness <u>modulo deadlocks</u> for $T_2 \in T_{imp+\Delta}(S)$ in (2). This connection explains why in correctness proofs deadlocking transformation schemes like $T_{imp+\Delta}$ are often desirable. For describing schedulers we are of course advised to use faithful transformation schemes only.

<u>Acknowledgement</u>. Research on this paper was supported by the German Research Council (DFG) under grant La 426/3-1.

References

/AO/ K.R. Apt, E.-R. Olderog, Proof rules and transformations dealing with fairness, Science of Computer Programming 3 (1983) 65-100; extended abstract appeared in: D. Kozen (Ed.), Proc. Logics of Programs 1981, Lecture Notes in Computer Science 131 (Springer, Berlin, 1982) 1-8.

/AP/ K.R. Apt, G.D. Plotkin, Countable nondeterminism and random assignment, Technical Report, Univ. of Edinburgh (1982); extended abstract appeared in: S. Even, O. Kariv (Eds.), Proc. 8th Coll. Automata, languages and Programming, Lecture Notes in Computer Science 115 (Springer, Berlin, 1981) 479-494.

/APS/ K.R. Apt, A. Pnueli, J. Stavi, Fair termination revisited - with delay, in: Proc. 2nd Conf. on Software Technology and Theoretical Computer Science, Bangalore (1982).

/Br1/ M. Broy, Transformational semantics for concurrent programs, Inform. Proc. Letters 11 (1980) 87-91.

/Br2/ M. Broy, Are fairness assumptions fair? , in: Proc. 2nd Intern. Conf. on Distributed Computing Systems (IEEE, Paris, 1981) 116-125.

/Di/ E.W. Dijkstra, A Discipline of Programming (Prentice Hall, 1976).

/FP/ M.J. Fisher, M.S. Paterson, Storage requirements for fair scheduling, Manuscript, Univ. of Warwick (1982).

/FS1/ L. Flon, N. Suzuki, Nondeterminism and the correctness of parallel programs, in: E.J. Neuhold (Ed.), Formal Description of Programming Concepts I (North Holland, Amsterdam, 1978) 589-608.

/FS2/ L. Flon, N. Suzuki, The total correctness of parallel programs, SIAM J. Comp. 10 (1981) 227-246.

/Fr/ N. Francez, Fairness, Unpublished Manuscript, Technion Univ. (1983).

/HP/ M.C.B. Henessy, G.D. Plotkin, Full abstraction for a simple programming language, in: J. Bečvář (Ed.), Proc. 8th Symp. on Mathematical Foundations of Computer Science 74 (Springer, Berlin, 1979) 108-120.

/Ho/ C.A.R. Hoare, Some properties of predicate transformers, J. ACM 25 (1978) 461-480.

/LPS/ D. Lehmann, A. Pnueli, J. Stavi, Impartiality, justice and fairness: the ethics of concurrent termination, in: S. Even, O. Kariv (Eds.), Proc. 8th Coll. Automata, Languages and Programming, Lecture Notes in Computer Science 115 (Springer, Berlin, 1981) 264-277.

/MP/ Z. Manna, A. Pnueli, Verification of concurrent programs: the temporal framework, in: R.S. Boyer, J.S. Moore (Eds.), The Correctness Problem in Computer Science, International Lecture Series in Computer Science (Academic Press, London, 1981).

/OG/ S. Owicki, D. Gries, An axiomatic proof technique for parallel programs, Acta Informatica 6 (1976) 319-340.

/OL/ S. Owicki, L. Lamport, Proving liveness properties of concurrent programs, ACM TOPLAS 4 (1982) 455-495.

/Pa/ D. Park, On the semantics of fair parallelism, in: D. Bjørner (Ed.), Proc. Abstract Software Specifications, Lecture Notes in Computer Science 86 (Springer, Berlin, 1979) 504-526.

/Pl/ G.D. Plotkin, A structural approach to operational semantics, Technical Report DAIMI-FN 19, Comp. Sci. Dept., Aarhus Univ. (1981).

/Pn/ A. Pnueli, The temporal semantics of concurrent programs, Theoretical Computer Science 13 (1981) 45-60.