Shared-state design modulo weak and strong process fairness

K. Stølen

Institut für Informatik, Technischen Universität München, Postfach 20 24 20, Arcisstrasse 21, D-8000 München 2, Germany

Abstract

A number of rules for proving shared-state programs totally correct with respect to both weak and strong process fairness is presented. The rules are compositional — thus they allow for design in a top-down style. On the other hand, already finished programs can be verified without the use of program transformation.

Keyword Codes: D.2.1; D.2.4; D.2.10
Keywords: Software Engineering, Requirements/Specifications; Program Verification; Design

1. INTRODUCTION

Appeared in: Proc. Forte'92, pages 479-498, North-Holland 1992.

In Hoare-logic a program is *totally correct* with respect to its pre/post specification iff it terminates in a state which satisfies the post-condition whenever the initial state satisfies the pre-condition. Rules for proving total correctness of shared-state programs with respect to different types of process fairness are proposed in [OA86]. The Olderog/Apt paper shows how both *unconditional*, *weak* and *strong* fairness can be reasoned about using *explicit scheduling*. The approach in [OA86] depends upon a freedom from interference test which can be carried out only after the component processes have been implemented and their proofs have been constructed. This is unacceptable when designing programs in a top-down style, because erroneous design decisions, taken early in the design process, may remain undetected until the whole program is complete. In the worst case, everything that depends upon such mistakes will have to be thrown away.

To avoid problems of this type a design method should satisfy what is known as the principle of *compositionality* [Zwi89] — namely that a specification of a program always can be verified on the basis of the specifications of its constituent components, without knowledge of the interior program structure of those components. Two other advantages of having a compositional method are: Firstly, specifications can be split into subspecifications which can be implemented separately (in isolation). Secondly, the specification of a component contains all information needed to use and combine it with other components.

The object of this paper is to present compositional rules for proving shared-state programs totally correct with respect to both weak and strong process fairness. The approach allows for design in a top-down style. On the other hand, already finished programs can be verified without the use of program transformation. In the style of [Jon83], [Stø90], [Stø91b], [XH91] *rely-* and *guarantee-conditions* are used to characterise interference. A *wait-condition* is employed to select

helpful paths. (The use of the wait-condition is to some degree related to the way [LPS81] and [GFMdR85] select helpful directions with respectively a state predicate and a rank. See [Fra86] for a detailed discussion of these and other approaches.)

A paper which proposes a similar set of rules for an unconditionally fair shared-state language has already been published [Stø92]. Since unconditional fairness is just a special case of weak fairness, the first part of the current paper follows to a large extent the presentation in [Stø92]. The basic programming language is first introduced. Then, weak fairness is defined and the corresponding formal system is presented. Based on the 'formal machinery' for the weakly fair case, it is then explained what it means for a language to be strongly fair, and it is shown how the deduction rules for weak fairness can be strengthened to handle strong fairness. The essential 'trick' is the use of a *prophecy variable* to predict the maximum number of states in which a process may be enabled without ever progressing.

2. PROGRAMMING LANGUAGE

Given that $\langle vl \rangle$, $\langle el \rangle$, $\langle tl \rangle$ and $\langle ts \rangle$ denote respectively a list of variables, a list of expressions, a list of types, and a Boolean test, then any program is of the form $\langle pg \rangle$, where

$$
\begin{array}{rcl}
\langle pg \rangle & ::= & \langle as \rangle \mid \langle bl \rangle \mid \langle sc \rangle \mid \langle wd \rangle \mid \langle pr \rangle \mid \langle aw \rangle \\
\langle as \rangle & ::= & \langle vl \rangle := \langle el \rangle \\
\langle bl \rangle & ::= & \mathsf{blo}\ \langle vl \rangle : \langle tl \rangle\ \mathsf{in}\ \langle pg \rangle\ \mathsf{olb} \\
\langle sc \rangle & ::= & \langle pg \rangle ; \langle pg \rangle \\
\langle wd \rangle & ::= & \mathsf{while}\ \langle ts \rangle\ \mathsf{do}\ \langle pg \rangle\ \mathsf{od} \\
\langle pr \rangle & ::= & \{ \langle pg \rangle \parallel \langle pg \rangle \} \\
\langle aw \rangle & ::= & \mathsf{await}\ \langle ts \rangle\ \mathsf{do}\ \langle pg \rangle\ \mathsf{od}
\end{array}
$$

Some readers may find it surprising that there is no if-statement. This is to save space. A rule for this construct can be found in [Stø92].

With respect to the assignment-statement $\langle as \rangle$, the two lists are required to correspond with respect to type and length, and the same variable is not allowed to occur in the variable list more than once.

The block-statement $\langle bl \rangle$ allows for the declaration of variables. Again, a one to one correspondence between the list of declared variables and the list of types is required. To avoid tedious complications due to name clashes it is assumed that for any program z, a variable x occurs in maximum one of z's declaration lists and only once in the same list. Moreover, if x occurs in the declaration list of one of z's block-statements z', then all occurrences of x in z are in z'. Strictly speaking, these (naming) assumptions break with the insistence upon compositionality, but as already pointed out: They are made only to simplify the presentation.

A program variable x is *local* with respect to a program z, if it occurs in a declaration list in z. Otherwise, x is *global* with respect to z. This means of course that a variable x which is local with respect to a program z, may be global with respect to some of z's subprograms.

The programming language is given operational semantics in the style of [Acz83]. To avoid complicating the semantics, while-, parallel- and await-statements are restricted from occurring in the body of an await-statement. (However, this constraint can easily be removed without having to modify any of the deduction rules.) A *state* is a mapping of all programming variables to values, while a *configuration* is a pair of the form $\langle z, s \rangle$, where z is a program or the *empty program* ϵ, and s is a state. The empty program ϵ models *termination*. s_ϑ denotes the state s

restricted to the set of variables ϑ, while $s \models b$ means that the Boolean expression b is true in the state s.

An *external* transition is the least binary relation on configurations such that

- $\langle z, s_1\rangle \xrightarrow{e} \langle z, s_2\rangle$,

while an *internal* transition is the least binary relation on configurations such that either

- $\langle v := r, s\rangle \xrightarrow{i} \langle \epsilon, s\binom{v}{r}\rangle$, where $s\binom{v}{r}$ denotes the state that is obtained from s, by mapping the variables v to the values of r in the state s, and leaving all other maplets unchanged,
- $\langle \mathsf{blo}\ v\colon T\ \mathsf{in}\ z\ \mathsf{olb}, s_1\rangle \xrightarrow{i} \langle z, s_2\rangle$, where s_2 denotes a state that is obtained from s_1, by mapping the variables in v to randomly chosen type-correct values, and leaving all other maplets unchanged,
- $\langle z_1; z_2, s_1\rangle \xrightarrow{i} \langle z_2, s_2\rangle$ if $\langle z_1, s_1\rangle \xrightarrow{i} \langle \epsilon, s_2\rangle$,
- $\langle z_1; z_2, s_1\rangle \xrightarrow{i} \langle z_3; z_2, s_2\rangle$ if $\langle z_1, s_1\rangle \xrightarrow{i} \langle z_3, s_2\rangle$ and $z_3 \neq \epsilon$,
- $\langle \mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od}, s\rangle \xrightarrow{i} \langle z; \mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od}, s\rangle$ if $s \models b$,
- $\langle \mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od}, s\rangle \xrightarrow{i} \langle \epsilon, s\rangle$ if $s \models \neg b$,
- $\langle \{z_1 \parallel z_2\}, s_1\rangle \xrightarrow{i} \langle z_2, s_2\rangle$ if $\langle z_1, s_1\rangle \xrightarrow{i} \langle \epsilon, s_2\rangle$,
- $\langle \{z_1 \parallel z_2\}, s_1\rangle \xrightarrow{i} \langle z_1, s_2\rangle$ if $\langle z_2, s_1\rangle \xrightarrow{i} \langle \epsilon, s_2\rangle$,
- $\langle \{z_1 \parallel z_2\}, s_1\rangle \xrightarrow{i} \langle \{z_3 \parallel z_2\}, s_2\rangle$ if $\langle z_1, s_1\rangle \xrightarrow{i} \langle z_3, s_2\rangle$ and $z_3 \neq \epsilon$,
- $\langle \{z_1 \parallel z_2\}, s_1\rangle \xrightarrow{i} \langle \{z_1 \parallel z_3\}, s_2\rangle$ if $\langle z_2, s_1\rangle \xrightarrow{i} \langle z_3, s_2\rangle$ and $z_3 \neq \epsilon$,
- $\langle \mathsf{await}\ b\ \mathsf{do}\ z_1\ \mathsf{od}, s_1\rangle \xrightarrow{i} \langle \epsilon, s_n\rangle$ if $s_1 \models b$ and there is a sequence of internal transitions such that
  - $\langle z_1, s_1\rangle \xrightarrow{i} \langle z_2, s_2\rangle \xrightarrow{i} \ldots \xrightarrow{i} \langle z_{n\text{-}1}, s_{n\text{-}1}\rangle \xrightarrow{i} \langle \epsilon, s_n\rangle$.

All functions occurring in expressions are required to be total. It follows from the definition that Boolean tests, assignment- and await-statements are atomic. In the rest of the paper skip will be used as an alias for the assignment of an empty list of expressions to an empty list of variables.

Definition 1 *A* computation *of a program z_1 is an infinite sequence of the form*

$$\langle z_1, s_1\rangle \xrightarrow{l_1} \langle z_2, s_2\rangle \xrightarrow{l_2} \ \ldots\ \xrightarrow{l_{k\text{-}1}} \langle z_k, s_k\rangle \xrightarrow{l_k} \ \ldots\ ,$$

where for all $j \geq 1$, $\langle z_j, s_j\rangle \xrightarrow{l_j} \langle z_{j+1}, s_{j+1}\rangle$ is either an external or an internal transition, and no external transition changes the values of z_1's local variables.

If σ is a computation of z, the idea is that an internal transition represents an atomic step due to z — in this paper often referred to as the *component*, while an external transition represents an atomic step due to z's *environment*, in other words, due to the other programs running in parallel with z. Note that an external transition can be performed in any configuration. Thus a computation may be infinite even if the component terminates. The constraint that no external transition changes the program's local variables can be built into the transition system by redefining configurations to have a third element — namely a set of local variables.

Given a computation σ, $Z(\sigma)$, $S(\sigma)$ and $L(\sigma)$ are the projection functions to sequences of programs, states and transition labels respectively, and for all $j \geq 1$, $Z(\sigma_j)$, $S(\sigma_j)$, $L(\sigma_j)$ and σ_j denote respectively the j'th program, the j'th state, the j'th transition label and the j'th configuration. $\sigma(j{:}\infty)$ denotes the result of removing the j-1 first transitions, while $\sigma(1{:}j)$ denotes the prefix of σ consisting of the j-1 first transitions. A configuration c is *disabled* iff $Z(c) \neq \epsilon$ and there is no configuration c', such that $c \xrightarrow{i} c'$. A program z is disabled in the state s iff the configuration $\langle z, s\rangle$ is disabled. A program is *enabled* in a state s if it is not disabled in s.

Two computations (or prefixes of computations) σ of z_1 and σ' of z_2 are *compatible*, if $\{z_1 \parallel z_2\}$ is a program, $S(\sigma) = S(\sigma')$ and for all $j \geq 1$, $L(\sigma_j) = L(\sigma'_j)$ implies $L(\sigma_j) = e$. More informally, σ of z_1 and σ' of z_2 are compatible, if there is no clash of variable names which restricts z_1 and z_2 from being composed in parallel, and for all n: The state in the n'th component of σ is equal to the state in the n'th component of σ'; if the n'th transition in σ is internal then the n'th transition in σ' is external; and if the n'th transition in σ' is internal then the n'th transition in σ is external. The reason for the two last constraints is of course that z_2 is a part of z_1's environment, and z_1 is a part of z_2's environment, thus an internal transition in σ must correspond to an external transition in σ', and the other way around.

For example, given three assignment-statements z_1, z_2 and z_3, the two computations

$$\langle z_1; z_2, s_1\rangle \xrightarrow{i} \langle z_2, s_2\rangle \xrightarrow{i} \langle \epsilon, s_3\rangle \xrightarrow{e} \sigma,$$
$$\langle z_3, s_1\rangle \xrightarrow{i} \langle \epsilon, s_2\rangle \xrightarrow{e} \langle \epsilon, s_3\rangle \xrightarrow{e} \sigma$$

are not compatible, because they both start with an internal transition. However,

$$\langle z_1; z_2, s_1\rangle \xrightarrow{i} \langle z_2, s_2\rangle \xrightarrow{e} \langle z_2, s_3\rangle \xrightarrow{e} \langle z_2, s_4\rangle \xrightarrow{i} \langle \epsilon, s_5\rangle \xrightarrow{e} \sigma,$$
$$\langle z_3, s_1\rangle \xrightarrow{e} \langle z_3, s_2\rangle \xrightarrow{i} \langle \epsilon, s_3\rangle \xrightarrow{e} \langle \epsilon, s_4\rangle \xrightarrow{e} \langle \epsilon, s_5\rangle \xrightarrow{e} \sigma$$

are compatible, and they can be *composed* into a unique computation

$$\langle \{z_1; z_2 \parallel z_3\}, s_1\rangle \xrightarrow{i} \langle \{z_2 \parallel z_3\}, s_2\rangle \xrightarrow{i} \langle z_2, s_3\rangle \xrightarrow{e} \langle z_2, s_4\rangle \xrightarrow{i} \langle \epsilon, s_5\rangle \xrightarrow{e} \sigma$$

of $\{z_1; z_2 \parallel z_3\}$, by composing the program part of each configuration, and making a transition internal iff one of the two component transitions are internal. More generally, for any pair of compatible computations σ and σ', let $\sigma \bowtie \sigma'$ denote

$$\langle z_1, s_1\rangle \xrightarrow{l_1} \langle z_2, s_2\rangle \xrightarrow{l_2} \ \ldots \ \xrightarrow{l_{k-1}} \langle z_k, s_k\rangle \xrightarrow{l_k} \ \ldots \ ,$$

where for all $j \geq 1$,

- $s_j = S(\sigma_j)$,
- $z_j = \{Z(\sigma_j) \parallel Z(\sigma'_j)\}$ if $Z(\sigma_j) \neq \epsilon$ and $Z(\sigma'_j) \neq \epsilon$,

- $z_j = Z(\sigma_j)$ if $Z(\sigma'_j) = \epsilon$, and $z_j = Z(\sigma'_j)$ if $Z(\sigma_j) = \epsilon$,
- $l_j = e$ iff $L(\sigma_j) = e$ and $L(\sigma'_j) = e$.

It is straightforward to show that:

Proposition 1 *For any pair of compatible computations σ of z_1 and σ' of z_2, $\sigma \bowtie \sigma'$ is uniquely determined by the definition above, and $\sigma \bowtie \sigma'$ is a computation of $\{z_1 \parallel z_2\}$.*

Proposition 2 *For any computation σ of $\{z_1 \parallel z_2\}$, there are two unique compatible computations σ' of z_1 and σ'' of z_2, such that $\sigma = \sigma' \bowtie \sigma''$.*

3. WEAK FAIRNESS

Informally, a computation is *weakly fair* iff each process, which becomes executable at some point in the computation, either terminates, performs infinitely many internal transitions or is disabled infinitely often. For example, the computation

$$\langle v := r, s_1\rangle \xrightarrow{e} \langle v := r, s_2\rangle \xrightarrow{e} \ldots \xrightarrow{e} \langle v := r, s_n\rangle \xrightarrow{e} \ldots$$

is not weakly fair since the assignment-statement $v := r$ is *infinitely overtaken* by the environment. Another example of a computation which is not weakly fair is

$$\langle \{z_1 \parallel z_2\}, s\rangle \xrightarrow{i} \langle \{z_1 \parallel \mathsf{skip}; z_2\}, s\rangle \xrightarrow{i} \ \ldots \ \xrightarrow{i} \langle \{z_1 \parallel z_2\}, s\rangle \xrightarrow{i} \langle \{z_1 \parallel \mathsf{skip}; z_2\}, s\rangle \xrightarrow{i} \ \ldots \ ,$$

where

z_1::
    $b := \mathsf{true}$,

z_2::
    while $\neg b$ do skip od.

The problem here is that z_1 is infinitely overtaken by z_2. Thus, weak fairness excludes both unfairness due to infinite overtaking by the overall environment, and unfairness which occurs when one component process is infinitely overtaken by another component process (see Section 7 for a more detailed discussion). Observe that only *top-level* weak fairness is considered here. In other words, nested parallel-statements are flattened as far as fairness is concerned.

To give a more formal definition, let $<$ be a binary relation on computations such that $\sigma < \sigma'$ iff $S(\sigma) = S(\sigma')$, $L(\sigma) = L(\sigma')$, and there is a (non-empty) program z such that for all $j \geq 1$, $Z(\sigma_j); z = Z(\sigma'_j)$. $\sigma \leq \sigma'$ means that $\sigma < \sigma'$ or $\sigma = \sigma'$. Clearly, for any computation σ, there is a minimal computation σ', such that $\sigma' \leq \sigma$ and for all computations σ'', if $\sigma'' < \sigma$ then $\sigma' \leq \sigma''$. Moreover, a computation σ is weakly fair iff its minimal computation σ' is weakly fair. For example a computation of the form

$$\langle z_1; z, s_1\rangle \xrightarrow{l_1} \langle z_2; z, s_2\rangle \xrightarrow{l_2} \ \ldots \ \xrightarrow{l_{j-1}} \langle z_j; z, s_j\rangle \xrightarrow{l_j} \ldots \ ,$$

where the subprogram z never becomes executable, is weakly fair if the computation

$$\langle z_1, s_1\rangle \xrightarrow{l_1} \langle z_2, s_2\rangle \xrightarrow{l_2} \ \ldots \ \xrightarrow{l_{j-1}} \langle z_j, s_j\rangle \xrightarrow{l_j} \ \ldots$$

is weakly fair.

It remains to state what it means for a minimal computation σ to be weakly fair. There are two cases: First of all, if there are two computations σ', σ'' and a $j \geq 1$, such that $Z(\sigma'_1) \neq \epsilon$,

$Z(\sigma_1'') \neq \epsilon$ and $\sigma' \bowtie \sigma'' = \sigma(j{:}\infty)$, then σ is weakly fair iff both σ' and σ'' are weakly fair. On the other hand, if σ cannot be decomposed in such a way, then σ is weakly fair iff σ terminates, σ has infinitely many internal transitions or σ has infinitely many configurations where the program component is disabled.

Definition 2 *Given a computation σ, if there is a computation σ', such that $\sigma' < \sigma$ then*

- *σ is weakly fair iff σ' is weakly fair,*

else if there are two computations σ', σ'' and a $j \geq 1$, such that $Z(\sigma_1') \neq \epsilon$, $Z(\sigma_1'') \neq \epsilon$ and $\sigma' \bowtie \sigma'' = \sigma(j{:}\infty)$ then

- *σ is weakly fair iff both σ' and σ'' are weakly fair,*

else

- *σ is weakly fair iff either*
  - *there is a $j \geq 1$, such that $Z(\sigma_j) = \epsilon$, or*
  - *for all $j \geq 1$, there is a $k \geq j$, such that $L(\sigma_k) = i$, or*
  - *for all $j \geq 1$, there is a $k \geq j$, such that σ_k is disabled.*

For example, given that the overall environment is restricted from changing the truth-value of b, then any weakly fair computation of $\mathsf{skip}; \{b{:=}\ \mathsf{true} \parallel \mathsf{while}\ \neg b\ \mathsf{do\ skip\ od}\}$ will terminate. To see this, firstly note that the first and third branch of the conditional in Definition 2 imply that the initial skip-statement eventually will be executed. But then it is clear that there is a $j > 1$ which allows for a decomposition according to the second branch, in which case it follows from the third branch that the assignment-statement eventually will be executed. This means that b eventually will become true. Thus, the assumption that the overall environment maintains the truth of b and the third branch imply that also the while-statement will terminate.

Observe that Propositions 1 and 2 also hold for weakly fair computations.

Definition 3 *Given a program z, let $cp_w(z)$ be the set of all weakly fair computations σ such that $Z(\sigma_1) = z$.*

A *specification* is an expression of the form $(\vartheta, \alpha){:}:(P, R, W, G, E)$, where ϑ is a finite set of programming variables, α is a finite set of auxiliary variables, the *pre-condition* P, and the *wait-condition* W are unary predicates, and the *rely-condition* R, the *guarantee-condition* G, and the *effect-condition* E are binary predicates. For any unary predicate U, $s \models U$ means that U is true in the state s. Moreover, for any binary predicate B, $(s_1, s_2) \models B$ means that B is true for the pair of states (s_1, s_2).

The *global state* is the state restricted to $\vartheta \cup \alpha$. It is required that $\vartheta \cap \alpha = \{\}$, and that P, R, W, G and E constrain only the variables in $\vartheta \cup \alpha$. This means for example, that if there are two states s, s', such that $s \models P$ and $s_{\vartheta \cup \alpha} = s'_{\vartheta \cup \alpha}$, then $s' \models P$.

In the case of binary predicates *hooked* variables (as in VDM [Jon90]) are employed to refer to the 'older' state. To avoid excessive use of parentheses it is assumed that $\Rightarrow$ has lower priority than $\wedge$ and $\vee$, which again have lower priority than $|$, which has lower priority than all other operator symbols. This means for example that $(a \wedge b) \Rightarrow c$ can be simplified to $a \wedge b \Rightarrow c$.

A specification states a number of *assumptions* about the environment. First of all, the initial state is assumed to satisfy the pre-condition. Secondly, it is assumed that any external transition, which changes the global state, satisfies the rely-condition. For example, given the rely-condition $x < \overleftarrow{x} \wedge y = \overleftarrow{y}$, it is assumed that the environment will never change the value of y. Moreover, if the environment assigns a new value to x, then this value will be less than or equal to the variable's previous value. The assumptions are summed up in the definition below:

Definition 4 *Given a set of variables ϑ, and pre- and rely-conditions P and R, then $ext(\vartheta, P, R)$ denotes the set of all computations σ, such that:*

- $S(\sigma_1) \models P$,
- *for all $j \geq 1$, if $L(\sigma_j) = e$ and $S(\sigma_j)_\vartheta \neq S(\sigma_{j+1})_\vartheta$ then $(S(\sigma_j), S(\sigma_{j+1})) \models R$.*

A specification is not only stating assumptions about the environment, but also *commitments* to the component. Given an environment which satisfies the assumptions, then the component is required either to (possibly busy) *wait* forever in states which satisfy the wait-condition or to *terminate*. Moreover, any internal transition, which changes the global state, is required to satisfy the guarantee-condition. Finally, if the component terminates, then the overall effect is constrained to satisfy the effect-condition. External transitions both before the first internal transition and after the last are included in the overall effect. This means that given the rely-condition $x > \overleftarrow{x}$, the strongest effect-condition for the program skip is $x \geq \overleftarrow{x}$. The commitments are summed up below:

Definition 5 *Given a set of variables ϑ, and wait-, guarantee- and effect-conditions W, G, E, then $int(\vartheta, W, G, E)$ denotes the set of all computations σ, such that:*

- *there is a $j \geq 1$, such that for all $k \geq j$, $S(\sigma_k) \models W$, or there is a $j \geq 1$, such that $Z(\sigma_j) = \epsilon$,*
- *for all $j \geq 1$, if $L(\sigma_j) = i$ and $S(\sigma_j)_\vartheta \neq S(\sigma_{j+1})_\vartheta$ then $(S(\sigma_j), S(\sigma_{j+1})) \models G$,*
- *for all $j \geq 1$, if $Z(\sigma_j) = \epsilon$ then $(S(\sigma_1), S(\sigma_j)) \models E$.*

A *specified program* is a pair consisting of a program z and a specification $(\vartheta, \alpha)::(P, R, W, G, E)$, written

$$z \text{ sat } (\vartheta, \alpha)::(P, R, W, G, E).$$

It is required that for any variable x occurring in z, x is an element of ϑ iff x is global with respect to z. Moreover, any variable occurring in z is restricted from being an element of α.

If the set of auxiliary variables is empty, it is now straightforward to characterise what it means for a specified program to be valid: Namely that any program computation which satisfies the environment assumptions, also satisfies the commitments to the component. More formally:

$$\models_w z \text{ sat } (\vartheta, \{\})::(P, R, W, G, E) \text{ iff } ext(\vartheta, P, R) \cap cp_w(z) \subseteq int(\vartheta, W, G, E).$$

Auxiliary variables are employed to increase the expressiveness. For example, without auxiliary variables many 'correct' developments are excluded because sufficiently strong intermediate predicates cannot be expressed.

In [OA86] the verification is conducted in several iterations. They first construct a proof in the style of ordinary Hoare-logic, then they prove freedom from interference and so on, and one way to make sure that the auxiliary structure remains the same from one iteration to the next is to implement the auxiliary structure in terms of program code. For the approach presented in this paper there is only one proof iteration and no need to 'remember' the auxiliary structure. This means that the use of auxiliary variables can be 'simulated' in the deduction rules. Observe that there are no constraints on the type of an auxiliary variable. For example the user is not restricted to reason in terms of full histories (history variables, traces etc.), but is instead free to define the auxiliary structure he prefers. This will now be explain in full detail.

If l and k are finite lists, then $\#l$ denotes the number of elements in l, $\langle l \rangle$ denotes the set of elements in l, $l \circ k$ denotes the result of prefixing k with l, while l_n, where $1 \leq n \leq \#l$, denotes the n'th element of l. Finally, $a \leftarrow \_(\vartheta, \alpha)u$ iff a is a list of variables, u is a list of expressions, and ϑ and α are two sets of variables, such that $\#a = \#u$, $\langle a \rangle \subseteq \alpha$, and for all $1 \leq j \leq \#a$, any variable occurring in u_j is an element of $\vartheta \cup \{a_j\}$.

An *augmentation* $\overset{(\vartheta,\alpha)}{\hookrightarrow}$, where ϑ and α are sets of variables, is the least binary relation on programs such that either

- $v := r \overset{(\vartheta,\alpha)}{\hookrightarrow} v \circ a := r \circ u$, where $a \leftarrow_{(\vartheta,\alpha)} u$,
- $\mathsf{blo}\ x\colon T\ \mathsf{in}\ z\ \mathsf{olb} \overset{(\vartheta,\alpha)}{\hookrightarrow} \mathsf{blo}\ x\colon T\ \mathsf{in}\ z'\ \mathsf{olb}$, where $z \overset{(\vartheta \cup \langle x \rangle,\alpha)}{\hookrightarrow} z'$,
- $z_1; z_2 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_1'; z_2'$, where $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_1'$ and $z_2 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2'$,
- $\mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od} \overset{(\vartheta,\alpha)}{\hookrightarrow} \mathsf{blo}\ b'\colon B\ \mathsf{in}\ b' \circ a := b \circ u; \mathsf{while}\ b'\ \mathsf{do}\ z'; b' \circ a := b \circ u\ \mathsf{od}\ \mathsf{olb}$, where $b' \notin \vartheta \cup \alpha$, $a \leftarrow_{(\vartheta,\alpha)} u$ and $z \overset{(\vartheta,\alpha)}{\hookrightarrow} z'$,
- $\{z_1 \parallel z_2\} \overset{(\vartheta,\alpha)}{\hookrightarrow} \{z_1' \parallel z_2'\}$, where $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_1'$ and $z_2 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2'$,
- $\mathsf{await}\ b\ \mathsf{do}\ z\ \mathsf{od} \overset{(\vartheta,\alpha)}{\hookrightarrow} \mathsf{await}\ b\ \mathsf{do}\ z'\ \mathsf{od}$, where $z \overset{(\vartheta,\alpha)}{\hookrightarrow} z'$.

The augmentation of an assignment statment allows a possibly empty list of auxiliary variables to be updated in the same atomic step as r is assigned to v. The additional constraint $a \leftarrow_{(\vartheta,\alpha)} u$ is needed to make sure that the elements of a really are auxiliary variables, and that two auxiliary variables do not depend upon each other. The latter requirement is necessary since it must be possible to remove some auxiliary variables from a specified program without having to remove all the auxiliary variables (see the elimination-rule at the end of the paper). The block-statement is used in the augmentation of the while-statement to allow auxiliary variables to be updated in the same atomic step as the Boolean test is evaluated.

It is now possible to define what it means for a specified program to be valid when the set of auxiliary variables is non-empty: Namely that the program can be augmented with auxiliary structure in such a way that any program computation which satisfies the environment assumptions, also satisfies the commitments to the component. More formally:

Definition 6 $\models_w z_1\ \underline{\mathrm{sat}}\ (\vartheta, \alpha)::(P, R, W, G, E)$ *iff there is a program* z_2, *such that* $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2$ *and* $ext(\vartheta \cup \alpha, P, R) \cap cp_w(z_2) \subseteq int(\vartheta \cup \alpha, W, G, E)$.

4. DECOMPOSITION RULES

The object of this section is to introduce the most important rules for the design of totally correct weakly fair programs. Some additional rules are listed at the end of the paper. This logic will be called LSP_w.

Given a list of expressions r, a set of variables ϑ, a unary predicate B, and two binary predicates C and D, then $\overleftarrow{r}$ denotes the list of expressions that can be obtained from r by hooking all free variables in r; $\overleftarrow{B}$ denotes a binary predicate such that $(s, s') \models \overleftarrow{B}$ iff $s \models B$; I_ϑ denotes the predicate $\bigwedge_{v \in \vartheta} v = \overleftarrow{v}$, while $C \mid D$ denotes the relational composition of C and D, in other words, $(s, s') \models C \mid D$ iff there is a state s'' such that $(s, s'') \models C$ and $(s'', s') \models D$. Moreover, C^+ denotes the transitive closure of C, while C^* denotes the reflexive and transitive closure of C. Finally, C is well-founded iff there is no infinite sequence of states $s_1, s_2, \ldots, s_n, \ldots$ such that for all $j \geq 1$, $(s_j, s_{j+1}) \models C$.

The *consequence-rule*

$$\frac{\begin{array}{l} P_2 \Rightarrow P_1 \\ R_2 \Rightarrow R_1 \\ W_1 \Rightarrow W_2 \\ G_1 \Rightarrow G_2 \\ E_1 \Rightarrow E_2 \\ z \ \underline{\mathrm{sat}}\ (\vartheta, \alpha) :: (P_1, R_1, W_1, G_1, E_1) \end{array}}{z \ \underline{\mathrm{sat}}\ (\vartheta, \alpha) :: (P_2, R_2, W_2, G_2, E_2)}$$

is straightforward. It basically states that it is sound to strengthen the assumptions and weaken the commitments.

The first version of the *assignment-rule*

$$\frac{\begin{array}{l} \overleftarrow{P} \wedge R \Rightarrow P \\ \overleftarrow{P} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \langle v \rangle} \Rightarrow (G \vee I_\vartheta) \wedge E \end{array}}{v := r \ \underline{\mathrm{sat}}\ (\vartheta, \{\}) :: (P, R, \mathsf{false}, G, R^* \mid E \mid R^*)}$$

can be used only when the set of auxiliary variables is empty. Any weakly fair computation is of the form

$$\langle v := r, s_1 \rangle \xrightarrow{e} \ \ldots \ \xrightarrow{e} \langle v := r, s_k \rangle \xrightarrow{i} \langle \epsilon, s_{k+1} \rangle \xrightarrow{e} \ \ldots \ \xrightarrow{e} \langle \epsilon, s_n \rangle \xrightarrow{e} \ \ldots \ .$$

Thus, the statement will always terminate, and there is only one internal transition. Moreover, since the initial state is assumed to satisfy P and any external transition, which changes the global state, is assumed to satisfy R, it follows from the first premise that $s_k \models P$ and that $(s_k, s_{k+1}) \models \overleftarrow{P} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \langle v \rangle}$. But then, it is clear from the second premise that $(s_k, s_{k+1}) \models G \vee I_\vartheta$, and that for all $l > k$, $(s_1, s_l) \models R^* \mid E \mid R^*$, which proves that the rule is sound.

In the general case, the execution of an assignment-statement $v := r$ corresponds to the execution of an assignment-statement of the form $v \circ a := r \circ u$, where $a \leftarrow_{(\vartheta, \alpha)} u$. Thus, the rule

$$\frac{\overleftarrow{P} \wedge R \Rightarrow P \quad \overleftarrow{P} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \langle v \rangle} \wedge a = \overleftarrow{u} \wedge I_{\alpha \setminus \langle a \rangle} \Rightarrow (G \vee I_{\vartheta \cup \alpha}) \wedge E \qquad a \leftarrow_{(\vartheta,\alpha)} u}{v := r \ \underline{\mathsf{sat}} \ (\vartheta, \alpha) :: (P, R, \mathsf{false}, G, R^* \mid E \mid R^*)}$$

is sufficient. The only real difference from above is that the second premise guarantees that the assignment-statement can be augmented with auxiliary structure in such a way that the specified changes to both the auxiliary variables and the programming variables will indeed take place.

Also the *while-rule*

$$\frac{\begin{array}{l}\overleftarrow{P} \wedge R \Rightarrow P \\ E^+ \wedge (R \vee G)^* \mid (I_\vartheta \wedge \neg W) \mid (R \vee G)^* \ \textit{is well-founded} \\ z \ \underline{\mathsf{sat}} \ (\vartheta, \{\,\}) :: (P \wedge b, R, W, G, P \wedge E)\end{array}}{\mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od}\ \underline{\mathsf{sat}}\ (\vartheta, \{\,\}) :: (P, R, W, G, R^* \mid (E^* \wedge \neg b) \mid R^*)}$$

is first given for the case that the set of auxiliary variables is empty. The unary predicate P can be thought of as an invariant which is true whenever the Boolean test b is evaluated. Since the conclusion's pre-condition restricts the initial state to satisfy P, and since it follows from the first premise that P is maintained by the environment, it follow that P is true when the Boolean test is evaluated for the first time. The occurrence of P in the effect-condition of the third premise implies that P is also true at any later evaluation of the Boolean test.

It follows from the third premise that the binary predicate E characterises the overall effect of executing the body of the while loop under the given environment assumptions. But then it is clear that the overall effect of the while-statement satisfies $R^* \mid (E^+ \wedge \neg b) \mid R^*$ if the loop iterates at least once, while the overall effect satisfies $R^* \mid (I_\vartheta \wedge \neg b) \mid R^*$ otherwise. This explains the conclusion's effect-condition. That any internal transition either leaves the state unchanged or satisfies G also follows from the third premise.

Note that $\wedge$ is the main symbol of the predicate in the second premise. To prove that the second premise implies that the statement terminates unless it ends up waiting in W, assume there is a non-terminating computation (a computation where the empty program is never reached)

$$\sigma \in ext(\vartheta, P, R) \cap cp_w(\mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od})$$

such that for all $j \geq 1$, there is a $k \geq j$, which satisfies $S(\sigma_k) \models \neg W$. It follows from the third premise that there is an infinite sequence of natural numbers $n_1 < n_2 < \ \ldots \ < n_k < \ \ldots$, such that for all $j \geq 1$,

$$(S(\sigma_{n_j}), S(\sigma_{n_{j+1}})) \models E.$$

But then, since by assumption $\neg W$ is true infinitely often, and since the overall effect of any finite sequence of external and internal transitions satisfies $(R \vee G)^*$, it follows that there is an infinite sequence of natural numbers $m_1 < m_2 < \ \ldots \ < m_k < \ \ldots$, such that for all $j \geq 1$,

$$(S(\sigma_{m_j}), S(\sigma_{m_{j+1}})) \models E^+ \wedge (R \vee G)^* \mid (I_\vartheta \wedge \neg W) \mid (R \vee G)^*.$$

This contradicts the second premise. Thus, the statement terminates or ends up waiting forever in W.

In the general case the following rule

$$\frac{\begin{array}{l}\overleftarrow{P_1} \wedge R \Rightarrow P_1 \\ (E_1 \mid E_2)^+ \wedge (R \vee G)^* \mid (I_{\vartheta\cup\alpha} \wedge \neg W) \mid (R \vee G)^* \textit{ is well-founded} \\ \mathsf{skip}\ \underline{\mathrm{sat}}\ (\vartheta,\alpha)::(P_1, \mathsf{false}, \mathsf{false}, G, P_2 \wedge E_1) \\ z\ \underline{\mathrm{sat}}\ (\vartheta,\alpha)::(P_2 \wedge b, R, W, G, P_1 \wedge E_2)\end{array}}{\mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od}\ \underline{\mathrm{sat}}\ (\vartheta,\alpha)::(P_1, R, W, G, R^* \mid (E_1 \mid E_2)^* \mid (E_1 \wedge \neg b) \mid R^*)}$$

is needed. Remember that any augmentation of a while-statement is of the form

$$\mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od} \overset{(\vartheta,\alpha)}{\hookrightarrow} \mathsf{blo}\ b'{:}B\ \mathsf{in}\ b' \circ a := b \circ u; \mathsf{while}\ b'\ \mathsf{do}\ z'; b' \circ a := b \circ u\ \mathsf{od}\ \mathsf{olb},$$

where $b' \notin \vartheta \cup \alpha$, $a \leftarrow_{(\vartheta,\alpha)} u$ and $z \overset{(\vartheta,\alpha)}{\hookrightarrow} z'$. The third premise simulates that auxiliary variables may be updated in the same atomic step as the Boolean test is evaluated. Moreover, this premise can be verified with the assignment-rule, since skip has been defined as an alias for the assignment of an empty list of expressions to an empty list of variables. Thus the overall effect is characterised by $R^* \mid (E_1 \mid E_2)^+ \mid (E_1 \wedge \neg b) \mid R^*$ if the body of the loop is executed at least once, and by $R^* \mid (E_1 \wedge \neg b) \mid R^*$ otherwise. This explains the conclusion's effect-condition.

It the case of the *parallel-rule*

$$\frac{\begin{array}{l}\neg(W_1 \wedge E_2) \wedge \neg(W_2 \wedge E_1) \wedge \neg(W_1 \wedge W_2) \\ z_1\ \underline{\mathrm{sat}}\ (\vartheta,\alpha)::(P, R \vee G_2, W \vee W_1, G_1, E_1) \\ z_2\ \underline{\mathrm{sat}}\ (\vartheta,\alpha)::(P, R \vee G_1, W \vee W_2, G_2, E_2)\end{array}}{\{z_1 \parallel z_2\}\ \underline{\mathrm{sat}}\ (\vartheta,\alpha)::(P, R, W, G_1 \vee G_2, E_1 \wedge E_2)}$$

note that the rely-condition of the second premise allows any interference due to z_2 (given the actual assumptions about the overall environment), and similarly that the rely-condition of the third premise allows any interference due to z_1. Thus since an effect-condition covers interference both before the first internal transition and after the last, it is clear from premise two and three that any internal transition, which changes the global state, satisfies $G_1 \vee G_2$, and that the overall effect satisfies $E_1 \wedge E_2$ if the program terminates.

It follows from the second premise that z_1 can end up waiting only in $W \vee W\_1$, when executed in an environment characterised by P and $R \vee G_2$. Moreover, the third premise implies that z_2 can end up waiting only in $W \vee W_2$, when executed in an environment characterised by P and $R \vee G\_1$. But then, since the first premise implies that z_1 cannot be waiting in W_1 after z_2 has terminated, that z_2 cannot be waiting in W_2 after z_1 has terminated, and that z_1 (z_2) cannot be waiting in $W_1 \wedge \neg W$ ($W_2 \wedge \neg W$) if z_2 (z_1) is waiting, it follows that $\{z_1 \parallel z_2\}$ is guaranteed to terminate or end up waiting in W if the overall environment is characterised by P and R.

To understand the *await-rule*

$$\frac{\begin{array}{l}\overleftarrow{P} \wedge R \Rightarrow P \\ (\overleftarrow{P} \wedge \overleftarrow{b} \wedge R \wedge \neg b) \mid R \textit{ is well-founded} \\ z\ \underline{\mathrm{sat}}\ (\vartheta,\alpha)::(P \wedge b, \mathsf{false}, \mathsf{false}, \mathsf{true}, (G \vee I_{\vartheta\cup\alpha}) \wedge E)\end{array}}{\mathsf{await}\ b\ \mathsf{do}\ z\ \mathsf{od}\ \underline{\mathrm{sat}}\ (\vartheta,\alpha)::(P, R, P \wedge \neg b, G, R^* \mid E \mid R^*)}$$

remember that the body of an await-statment is atomic, and also that it has no occurrences of while-, parallel- or await-statements. The atomicity explains the rely- and wait-condition of the third premise. Since the initial state is restricted to satisfy P, it follows from the first premise

that P remains true at least until the execution of the await-statment's body is started up. But, then since the second premise implies that the environment cannot change the truth-value of b more than a finite number of times, it follows that the statement either terminates or ends up waiting in $P \wedge \neg b$. Moreover the third premise implies that any atomic change of state satisfies G, and that the overall effect is characterised by $R^* \mid E \mid R^*$.

As an example, assume again that

z_1::
    b:= true,

z_2::
    while $\neg b$ do skip od.

then clearly

$$\models_w \{z_1 \parallel z_2\} \ \underline{\text{sat}} \ (\{b\}, \{\}) :: (\text{true}, \overleftarrow{b} \Rightarrow b, \text{false}, \text{true}, \text{true}).$$

This follows easily by the consequence- and parallel-rules, if

$$\vdash_w z_1 \ \underline{\text{sat}} \ (\{b\}, \{\}) :: (\text{true}, \overleftarrow{b} \Rightarrow b, \text{false}, \overleftarrow{b} \Rightarrow b, b),$$
$$\vdash_w z_2 \ \underline{\text{sat}} \ (\{b\}, \{\}) :: (\text{true}, \overleftarrow{b} \Rightarrow b, \neg b, \overleftarrow{b} \Rightarrow b, \text{true}).$$

(For any specified program ψ, $\vdash_w \psi$ iff ψ is provable in LSP$_w$.) The first of these specified programs can be deduced by the consequence- and assignment-rule. The second follows by the consequence- and while-rule, since

$$\vdash_w \text{skip} \ \underline{\text{sat}} \ (\{b\}, \{\}) :: (\neg b, \overleftarrow{b} \Rightarrow b, \neg b, \overleftarrow{b} \Rightarrow b, \neg \overleftarrow{b}),$$

and it is clear that $\neg \overleftarrow{b} \wedge (\overleftarrow{b} \Rightarrow b) \mid ((\overleftarrow{b} \Leftrightarrow b) \wedge b) \mid (\overleftarrow{b} \Rightarrow b)$ is well-founded.

5. STRONG FAIRNESS

The object of this section is to explain what it means for a computation to be *strongly fair*. Informally, a computation is strongly fair iff each process which becomes executable at some point in the computation, either terminates, performs infinitely many internal transitions, or is continuously disabled from a certain point onwards. For example, if

z_1::
    await $a \vee \neg b$ do a:= true od,

z_2::
    await $a \vee b$ do a:= true od,

then any strongly fair computation of $\{z_1 \parallel z_2\}$ terminates, given that the overall environment satisfies the rely-condition $\overleftarrow{a} \Rightarrow a$. However, under the same environment assumption, there are weakly fair computations of $\{z_1 \parallel z_2\}$ which does not terminate. If for all $j \geq 1$, $Z(\sigma_j) = \{z_1 \parallel z_2\}$, $L(\sigma_j) = e$, $S(\sigma_j) \models \neg a \wedge b$ if $j \bmod 2 = 1$, and $S(\sigma_j) \models \neg a \wedge \neg b$ if $j \bmod 2 = 0$, then σ is a weakly fair nonterminating computation of $\{z_1 \parallel z_2\}$.

A formal definition of strong fairness is given below:

Definition 7 *Given a computation σ, if there is a computation σ', such that $\sigma' < \sigma$ then*

- *σ is strongly fair iff σ' is strongly fair,*

else if there are two computations σ', σ'' and a $j \geq 1$, such that $Z(\sigma'_1) \neq \epsilon$, $Z(\sigma''_1) \neq \epsilon$ and $\sigma' \bowtie \sigma'' = \sigma(j{:}\infty)$ then

- *σ is strongly fair iff both σ' and σ'' are strongly fair,*

else

- *σ is strongly fair iff either*
  - *there is a $j \geq 1$, such that $Z(\sigma_j) = \epsilon$, or*
  - *for all $j \geq 1$, there is a $k \geq j$, such that $L(\sigma_k) = i$, or*
  - *there is a $j \geq 1$, such that for all $k \geq j$, σ_k is disabled.*

The only 'real' change with respect to the definition of weak fairness is in the last line, which restricts a process which has not yet terminated to eventually progress unless it is continuously disabled from a certain point onwards. Propositions 1 and 2 also hold for strongly fair computations.

Definition 8 *Given a program z, let $cp_s(z)$ be the set of all strongly fair computations σ, such that $Z(\sigma_1) = z$.*

6. DECOMPOSITION RULES

A specified program is defined in exactly the same way as earlier. Moreover, the definition of validity is simply:

Definition 9 *$\models_s\ z_1\ \underline{\text{sat}}\ (\vartheta, \alpha)::(P, R, W, G, E)$ iff there is a program z_2, such that $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2$ and $ext(\vartheta \cup \alpha, P, R) \cap cp_s(z_2) \subseteq int(\vartheta \cup \alpha, W, G, E)$.*

What remains is to update the deduction rules. This new logic is called LSP_s, while $\vdash_s$ is used to denote that a specified program is provable in LSP_s. Actually, the only rule which needs to be changed is the await-rule. In the case of strong fairness the second premise is no longer needed:

$$\frac{\overleftarrow{P} \wedge R \Rightarrow P \qquad z\ \underline{\text{sat}}\ (\vartheta, \alpha)::(P \wedge b, \mathsf{false}, \mathsf{false}, \mathsf{true}, (G \vee I_{\vartheta\cup\alpha}) \wedge E)}{\mathsf{await}\ b\ \mathsf{do}\ z\ \mathsf{od}\ \underline{\text{sat}}\ (\vartheta, \alpha)::(P, R, P \wedge \neg b, G, R^* \mid E \mid R^*)}$$

The rule is still sound because if the environment changes the truth-value of b infinitely many times, then the body of the statement will eventually be executed.

Given this rule, if

z_1::
  await $a \vee \neg b$ do $a := \mathsf{true}$ od,

z_2::
  await $a \vee b$ do $a := \mathsf{true}$ od,

then

$$\vdash_s \{z_1 \parallel z_2\}\ \underline{\text{sat}}\ (\{a, b\}, \{\})::(\mathsf{true}, \overleftarrow{a} \Rightarrow a, \mathsf{false}, \mathsf{true}, \mathsf{true}).$$

follows by the consequence- and parallel-rule, since it is straightforward to show that

$$\vdash_s z_1\ \underline{\text{sat}}\ (\{a, b\}, \{\})::(\mathsf{true}, \overleftarrow{a} \Rightarrow a, \neg a \wedge b, \overleftarrow{a} \Rightarrow a, a),$$
$$\vdash_s z_2\ \underline{\text{sat}}\ (\{a, b\}, \{\})::(\mathsf{true}, \overleftarrow{a} \Rightarrow a, \neg a \wedge \neg b, \overleftarrow{a} \Rightarrow a, a).$$

Nevertheless, a stronger logic is needed. Let

z_1::
await a do b:= true od,

z_2::
while $\neg b$ do a:= false; a:= true od,

then it should be clear that

$$\models_s \{z_1 \parallel z_2\} \underline{\text{sat}} (\{a, b\}, \{\}) :: (\text{true}, \text{false}, \text{false}, \text{true}, \text{true}).$$

Unfortunately, given the rules already introduced, this cannot be proved. Although it is easy to prove that

$$\vdash_s z_1 \underline{\text{sat}} (\{a, b\}, \{\}) :: (\text{true}, \overleftarrow{b} \Rightarrow b, \neg a, (\overleftarrow{a} \Rightarrow a) \wedge (\overleftarrow{b} \Rightarrow b), b),$$
$$\vdash_s z_2 \underline{\text{sat}} (\{a, b\}, \{\}) :: (\text{true}, (\overleftarrow{a} \Rightarrow a) \wedge (\overleftarrow{b} \Rightarrow b), \neg b, \overleftarrow{b} \Rightarrow b, a),$$

it does not follow from this and the parallel-rule that $\{z_1 \parallel z_2\}$ terminates. The reason is of course that the conjunction of the two wait-conditions is no contradiction.

The obvious question at this point is: In what way may LSP_s be strengthened to get around this incompleteness? The solution is to employ a *prophecy variable*. Prophecy variables make predictions about the future in the same way as ordinary auxiliary variables record the past. Prophecy variables were first used in [AL88]. In LSP_s only one prophecy variable, syntactically distinguished as $\wp$, is needed. Actually, since $\wp$ remains constant during the whole computation, *prophecy constant* is perhaps a better name.

An await-statement z *deadlocks* in a computation σ iff:

- if there is a computation σ', such that $\sigma' < \sigma$ then
  - z deadlocks in σ iff z deadlocks in σ',

  else if there are two computations σ', σ'' and a $j \geq 1$, such that $Z(\sigma'_1) \neq \epsilon$, $Z(\sigma''_1) \neq \epsilon$ and $\sigma' \bowtie \sigma'' = \sigma(j{:}\infty)$ then
  - z deadlocks in σ iff z deadlocks in σ' or in σ'',

  else
  - z deadlocks in σ iff there is a $j \geq 1$, such that $Z(\sigma_j) = z$ and for all $k \geq j$, σ_k is disabled.

The idea is that for any computation σ the initial value of $\wp$ is assumed to be greater than or equal to the maximum number of states in which an await-statement, which deadlocks in σ, is enabled in σ. Clearly since a computation can have only a finite number of deadlocked await-statements, and it follows from the fairness constraint that no such statement can be enabled in more than a finite number of states, it follows that for any computation such a maximum number exists.

On the other hand, there is no such maximum number for all computations. Thus the value of $\wp$ will vary from one computation to the next. This means that $\wp$ is no ordinary constant, but should instead be thought of as a variable which makes a prediction about the future. In some sense, the nondeterministic initialisation of $\wp$ can be understood as a way of moving forward in time some of the nondeterministic decisions taken at different points in the computation.

Before showing how this prophecy variable makes it possible to overcome the problem above, it is necessary to point out some restrictions. First of all, as already mentioned, the value of $\wp$ remains constant during the whole computation, and $\wp$ is of course not allowed to occur in programs. Moreover, to make sure that $\wp$ is not misused, in specifications $\wp$ is restricted to occur in wait-conditions only. This means for example that the user can never use the pre-condition to constrain $\wp$'s initial value. However, whenever $\wp$ occurs in a predicate it may be treated and reasoned about as an ordinary constant.

Given these restrictions, LSP_s may be strengthened with the following *prophecy-rule*:

$$\frac{\begin{array}{l} P \Rightarrow k = 0 \\ R \Rightarrow k = \overleftarrow{k} \vee (k = \overleftarrow{k} + 1 \wedge b \wedge \neg \overleftarrow{b}) \\ \mathsf{await}\ b\ \mathsf{do}\ z\ \mathsf{od}\ \underline{\mathrm{sat}}\ (\vartheta, \alpha)::(P, R, W, G, E) \end{array}}{\mathsf{await}\ b\ \mathsf{do}\ z\ \mathsf{od}\ \underline{\mathrm{sat}}\ (\vartheta, \alpha)::(P, R, W \wedge k \leq \wp, G, E)}$$

To see that this rule is sound, first observe that if σ is a non-terminating computation of an await-statement, then σ is of the form $\sigma_1 \xrightarrow{e} \sigma_2 \xrightarrow{e} \ldots \xrightarrow{e} \sigma_n \xrightarrow{e} \ldots$. In other words, σ has only external transitions and $Z(\sigma_1)$ deadlocks. (Remember that while-, await- and parallel-statements have been constrained from occurring in the body of an await-statement.) It follows from the first premise that the value of k in the initial state $S(\sigma_1)$ is 0. The second premise implies that an external transition either leaves the value of k unchanged or increments the value of k with 1, with the additional constraint that k is left unchanged unless the truth-value of b is changed from false to true in the same atomic step. This means that for any $j \geq 1$, the value of k in the configuration σ_j is less than or equal to the number of enabled states in $\sigma(1{:}\,j)$. Moreover, since σ is non-terminating, it follows from the fairness constraint that there are only finitely many enabled configurations in σ. Thus, from a certain point onwards the value of k remains constant, which means that there is a natural number denoted by $\wp$ such that for any $j \geq 1$, the value of k in σ_j is less than or equal to $\wp$. This and the third premise imply the conclusion.

Let once more

$z_1::$ $\mathsf{await}\ a\ \mathsf{do}\ b := \mathsf{true}\ \mathsf{od},$ $\qquad$ $z_2::$ $\mathsf{while}\ \neg b\ \mathsf{do}\ a := \mathsf{false};\ a := \mathsf{true}\ \mathsf{od},$

then

$$\vdash_s \{z_1 \parallel z_2\}\ \underline{\mathrm{sat}}\ (\{a, b\}, \{\})::(\mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{true}, \mathsf{true})$$

follows by the consequence- and elimination-rule (see rules at the end of the paper) if

$$\vdash_s \{z_1 \parallel z_2\}\ \underline{\mathrm{sat}}\ (\{a, b\}, \{k\})::(k = 0, I_{\{k,a,b\}}, \mathsf{false}, \mathsf{true}, \mathsf{true}).$$

The latter follows by the consequence- and parallel-rule if

$$\vdash_s z_1\ \underline{\mathrm{sat}}\ (\{a, b\}, \{k\})::(k = 0, G_2, \neg a \wedge k \leq \wp, G_1, b),$$
$$\vdash_s z_2\ \underline{\mathrm{sat}}\ (\{a, b\}, \{k\})::(k = 0, G_1, \neg b \wedge k > \wp, G_2, a)$$

where

$$G_1 \stackrel{\mathsf{def}}{=} (\overleftarrow{a} \Rightarrow a) \wedge (\overleftarrow{b} \Rightarrow b) \wedge k = \overleftarrow{k},$$
$$G_2 \stackrel{\mathsf{def}}{=} (\overleftarrow{b} \Rightarrow b) \wedge (k = \overleftarrow{k} \vee (k = \overleftarrow{k} + 1 \wedge a \wedge \neg \overleftarrow{a})).$$

Due to the prophecy-rule it is now straightforward to prove that z_1 satisfies its specification. Moreover, since it can be easily shown that (remember that k is auxiliary)

$$\vdash_s a := \mathsf{false}; a := \mathsf{true} \ \underline{\mathrm{sat}}$$
$$(\{a, b\}, \{k\}) :: (\neg b, G_1, \neg b \wedge k > \wp, G_2, \neg \overleftarrow{b} \wedge a \wedge k = \overleftarrow{k} + 1)$$

and the predicate $(\neg \overleftarrow{b} \wedge a \wedge k = \overleftarrow{k} + 1) \wedge (G_1 \vee G_2)^* \mid (I_{\{a,b,k\}} \wedge (b \vee k \leq \wp)) \mid (G_1 \vee G_2)^*$ is well-founded, it follows by the consequence- and while-rule that z_2 also satisfies its specification.

7. DISCUSSION

With respect to weak fairness, observe that Definition 2 not only constrains the programming language's parallel construct to be weakly fair, but also restricts the component from being *infinitely overtaken* by the overall environment. The latter restriction can be thought of as an assumption about the environment built into the semantics. It may be argued that it would have been more correct to state this assumption at the specification level as an additional assumption in definition 4. In that case, the definition of weak fairness can be weakened to allow for infinite overtaking by the overall environment. However, this distinction is not of any great practical importance since the deduction rules are exactly the same for both interpretations. The same is true for the definition of strong fairness.

Moreover, it is important to realise that the method presented here is not intended as a tool for writing and reasoning about interface specifications with respect to general environments, as in for example [AL90]. Instead the object of this paper is to propose a number of rules which allows for top-down design/verification of programs in a certain programming language, and the reason why specifications are written on an assumption/commitment form is to ensure compositionality of the design rules. Thus at any point it may be assumed that the environment is either empty or another subprocess of the program currently under design — in which case the environment restrictions imposed by the fairness constraints are not too strong.

Examples where logics related to LSP_w and LSP_s are used for the development of non-trivial programs can be found in [Stø90], [Stø91a]. [Stø90], [Stø91b] present a system called LSP which has been proved semantically (relatively) complete with respect to an unfair language. [Stø92] describes a similar formalism called LSP_u which deals with an unconditionally fair language. This set of rules has also been proved semantically (relatively) complete. The only difference between LSP_u and LSP_w is that the latter has an await-rule. This is as expected, since unconditional and weak fairness coincides if there is no await-statement. Thus, it should be fairly easy to extend the semantic completeness proof of LSP_u to LSP_w.

8. ACKNOWLEDGEMENTS

I am indebted to Howard Barringer, Cliff B. Jones, Mathai Joseph and Xu Qiwen.

References

[Acz83] P. Aczel. On an inference rule for parallel composition. Unpublished Paper, February 1983.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. Technical Report 29, Digital, Palo Alto, 1988.

[AL90] M. Abadi and L. Lamport. Composing specifications. Technical Report 66, Digital, Palo Alto, 1990.

[Fra86] N. Francez. *Fairness.* Springer, 1986.

[GFMdR85] O. Grumberg, N. Francez, J.A. Makowsky, and W. P. de Roever. A proof rule for fair termination of guarded commands. *Information and Control*, 66:83–102, 1985.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In Mason R.E.A., editor, *Proc. Information Processing 83*, pages 321–331. North-Holland, 1983.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM, Second Edition.* Prentice-Hall, 1990.

[LPS81] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proc. Automata, Languages, and Programming, Lecture Notes in Computer Science 115*, pages 264–277, 1981.

[OA86] E. R. Olderog and K. R. Apt. Fairness in parallel programs: The transformational approach. Technical Report 86-1, Liens, 1986.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures.* PhD thesis, University of Manchester, 1990. Also available as technical report UMCS-91-1-1, University of Manchester.

[Stø91a] K. Stølen. An attempt to reason about shared-state concurrency in the style of VDM. In S. Prehn and W. J. Toetenel, editors, *Proc. VDM'91, Lecture Notes in Computer Science 552*, pages 324–342, 1991.

[Stø91b] K. Stølen. A method for the development of totally correct shared-state parallel programs. In J. C. M. Baeten and J. F. Groote, editors, *Proc. CONCUR'91, Lecture Notes in Computer Science 527*, pages 510–525, 1991.

[Stø92] K. Stølen. Proving total correctness with respect to a fair (shared-state) parallel language. In C. B. Jones and R. C. Shaw, editors, *Proc. 5th. BCS-FACS Refinement Workshop*, pages 320–341. Springer, 1992.

[XH91] Q. Xu and J. He. A theory of state-based parallel programming by refinement:part 1. In J. M. Morris and R. C. Shaw, editors, *Proc. 4th BCS-FACS Refinement Workshop.* Springer, 1991.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness: Proof Theories for Networks of Processes and Their Relationship*, volume 321 of *Lecture Notes in Computer Science.* Springer, 1989.

block::

$$\frac{z \ \underline{\text{sat}} \ (\vartheta, \alpha)::(P, R \wedge x = \overleftarrow{x}, W, G, E)}{\text{blo } x\text{: } T \text{ in } z \text{ olb } \underline{\text{sat}} \ (\vartheta \setminus \langle x \rangle, \alpha)::(P, R, W, G, E)}$$

pre: :

$$\frac{z \ \underline{\text{sat}} \ (\vartheta, \alpha) : : (P, R, W, G, E)}{z \ \underline{\text{sat}} \ (\vartheta, \alpha) : : (P, R, W, G, \overleftarrow{P} \wedge E)}$$

elimination: :

$$\frac{x \notin \vartheta \qquad z \ \underline{\text{sat}} \ (\vartheta, \alpha) : : (P, R, W, G, E)}{z \ \underline{\text{sat}} \ (\vartheta, \alpha \setminus \{x\}) : : (\exists x.P, \forall \overleftarrow{x} .\exists x.R, W, G, E)}$$

sequential: :

$$\frac{z_1 \ \underline{\text{sat}} \ (\vartheta, \alpha) : : (P_1, R, W, G, P_2 \wedge E_1) \qquad z_2 \ \underline{\text{sat}} \ (\vartheta, \alpha) : : (P_2, R, W, G, E_2)}{z_1; z_2 \ \underline{\text{sat}} \ (\vartheta, \alpha) : : (P_1, R, W, G, E_1 \mid E_2)}$$