Proving Total Correctness with Respect to a Fair (Shared-State) Parallel Language

Ketil Stølen[*]
Institut für Informatik, der Technischen Universität,
Munich, Germany

Abstract

A method for proving programs totally correct with respect to an unconditionally fair (shared-state) parallel language is presented. The method is compositional and well-suited for top-down program development. It does not depend upon temporal logic, and program transformation is not employed. A number of examples are given.

1 Introduction

Appeared in: Proc. 5'th Refinement Workshop, pages 320-341, Springer 1992.

The two most well-known approaches to proving fair termination are the *explicit schedulers* method and the *helpful directions* method. The explicit schedulers method [Par81] transforms a fair termination problem into an ordinary non-deterministic termination problem by augmenting the code with statements in such a way that the new program simulates exactly all fair computations of the old program.

The helpful directions method is based upon choosing helpful directions at intermediate stages. There are two variants of this approach. One bases the selection upon a rank, where each rank has its own set of helpful directions [GFMdR85], while the other employs a state predicate to determine the helpful directions [LPS81]. A third way of reasoning about fairness is to use temporal logic, as in for example [GPSS80].

The approach described in this paper does not employ any of the techniques mentioned above, although it is to some degree related to the helpful directions method. In the style of LSP [Stø90], [Stø91a], [Stø91b] rely- and guarantee-conditions are used to characterise interference, while a wait-condition is employed to select helpful paths. Auxiliary variables are needed. However, they are not first implemented and thereafter removed, as for example in the Owicki/Gries method [OG76]. Instead the use of auxiliary variables is 'simulated' in the deduction rules.

The next section, Section 2, defines the basic programming language. The syntax and meaning of specifications are the topics of Section 3, while the most

[*]Author's address: Institute für Informatik, der Technischen Universität, Postfach 20 24 20, Arcisstrasse 21, D-8000 München 2, Germany. Email address: stoelen@informatik.tu-muenchen.de

important deduction-rules are explained in Section 4. Section 5 consists of a number of examples. Finally, Section 6 indicates some extensions and compares the method to other methods known from the literature.

2 Programming Language

The object of this section is to characterise the programming language, both at syntactic and semantic level. A *program*'s context-independent syntax is characterised in the well-known BNF-notation: given that $\langle vl \rangle$, $\langle el \rangle$, $\langle dl \rangle$ and $\langle ts \rangle$ denote respectively a list of variables, a list of expressions, a list of variable declarations, and a Boolean test, then any program is of the form $\langle pg \rangle$, where

$$\begin{aligned}
\langle pg \rangle &::= \langle as \rangle \mid \langle bl \rangle \mid \langle sc \rangle \mid \langle if \rangle \mid \langle wd \rangle \mid \langle pr \rangle \\
\langle as \rangle &::= \langle vl \rangle := \langle el \rangle \\
\langle bl \rangle &::= \mathsf{blo}\ \langle dl \rangle\ \mathsf{in}\ \langle pg \rangle\ \mathsf{olb} \\
\langle sc \rangle &::= \langle pg \rangle ; \langle pg \rangle \\
\langle if \rangle &::= \mathsf{if}\ \langle ts \rangle\ \mathsf{then}\ \langle pg \rangle\ \mathsf{else}\ \langle pg \rangle\ \mathsf{fi} \\
\langle wd \rangle &::= \mathsf{while}\ \langle ts \rangle\ \mathsf{do}\ \langle pg \rangle\ \mathsf{od} \\
\langle pr \rangle &::= \{\langle pg \rangle \parallel \langle pg \rangle\}
\end{aligned}$$

The main structure of a program is characterised above. However, a syntactically correct program is also required to satisfy some supplementary constraints. First of all, with respect to the assignment-statement $\langle as \rangle$, it is required that the two lists have the same number of elements, that the j'th variable in the first list is of the same type as the j'th expression in the second, and that the same variable does not occur in the variable list more than once.

The block-statement $\langle bl \rangle$ allows for the declaration of variables $\langle dl \rangle$. To avoid tedious complications due to name clashes: for any program z, a variable x may occur 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'.

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.

In LSP [Stø90] variables occurring in the Boolean test of an if- or a while-statement are constrained from being accessed by any other program running in parallel. In this paper there is no such constraint.

The programming language is given operational semantics in the style of [Acz83]. 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. Moreover, 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

- $\langle v{:=}\ r, s\rangle \xrightarrow{i} \langle \epsilon, s(^v_r)\rangle$, where $s(^v_r)$ 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,

- $\langle \mathsf{blo}\ d\ \mathsf{in}\ z\ \mathsf{olb}, s_1\rangle \xrightarrow{i} \langle z, s_2\rangle$, where s_2 denotes a state that is obtained from s_1, by mapping the variables in d 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 $z_3 \neq \epsilon$,

- $\langle \mathsf{if}\ b\ \mathsf{then}\ z_1\ \mathsf{else}\ z_2\ \mathsf{fi}, s\rangle \xrightarrow{i} \langle z_1, s\rangle$ if $s \models b$,

- $\langle \mathsf{if}\ b\ \mathsf{then}\ z_1\ \mathsf{else}\ z_2\ \mathsf{fi}, s\rangle \xrightarrow{i} \langle z_2, s\rangle$ if $s \models \neg b$,

- $\langle \mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od}, s\rangle \xrightarrow{i} \langle z; \mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od}, s\rangle$ 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 $z_3 \neq \epsilon$.

The above definition is of course sensible only if expressions in assignment-statements and Boolean tests never evaluate to 'undefined'. Thus, all functions occurring in expressions must be required to be total.

It follows from the definition that Boolean tests and assignment-statements are atomic. The empty program ϵ models *termination.*

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*

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

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, 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.

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. $\sigma(j, \dots, \infty)$ denotes the result of removing the j-1 first transitions.

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 (see restriction on local variable names above), 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

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

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

It is straightforward to show that:

Statement 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 $\{z_1 \parallel z_2\}$.*

Moreover, it is also easy to prove that:

Statement 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 $\sigma = \sigma' \bowtie \sigma''$.*

So far no fairness constraint has been introduced. This means for example that if z_1 and z_2 denote respectively the programs

$$b := \mathsf{true}, \qquad \mathsf{while}\ \neg b\ \mathsf{do\ skip\ od},$$

then there is no guarantee that $\{z_1 \parallel z_2\}$ terminates even if the program is executed in an environment which is restricted from changing the truth-value of b. There are two reasons for this:

- $\{z_1 \parallel z_2\}$ may be infinitely overtaken by the environment, as for example in the computation:

  $$\langle\{z_1 \parallel z_2\}, s_1\rangle \xrightarrow{e} \langle\{z_1 \parallel z_2\}, s_2\rangle \xrightarrow{e} \ \ldots \ \xrightarrow{e} \langle\{z_1 \parallel z_2\}, s_j\rangle \xrightarrow{e} \ \ldots\ .$$

  In this particular computation all transitions are external. However, more generally, a computation suffers from this type of unfairness if it has only finitely many internal transitions and the empty program is never reached (it does not terminate).

- z_1 may be infinitely overtaken by z_2. This is for example the case in the computation:

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

  In the general case, a computation which suffers from this type of unfairness, has infinitely many internal transitions. Moreover, from a certain point, at least one *process* is infinitely overtaken.

Informally, a computation is *unconditionally fair* iff each process, which becomes executable at some point in the computation, either terminates or performs infinitely many internal transitions. Observe that this excludes both unfairness due to infinite overtaking by the overall environment, and unfairness which occurs when one process is infinitely overtaken by another process. (What is called unconditional fairness in this paper is inspired by the definition of impartiality in [LPS81].)

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 $j \geq 1$, $Z(\sigma_j); z = Z(\sigma'_j)$. Moreover, $\sigma \leq \sigma'$ 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 unconditionally fair iff its minimal computation σ' is unconditionally 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 unconditionally 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 unconditionally fair.

It remains to state what it means for a minimal computation σ to be unconditionally 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 $\sigma' \bowtie \sigma'' = \sigma(j, \ldots, \infty)$, then σ is unconditionally fair iff both σ' and σ'' are unconditionally fair. On the other hand, if σ cannot be decomposed in such a way, then σ is unconditionally fair iff σ terminates or σ has infinitely many internal transitions.

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

- *σ is unconditionally fair iff σ' is unconditionally 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 $\sigma' \bowtie \sigma'' = \sigma(j, \ldots, \infty)$ then

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

else

- *σ is unconditionally 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 $L(\sigma_k) = i$.*

Observe that propositions 1 and 2 also hold for unconditionally fair computations.

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

The definition above not only constrains the programming language's parallel construct to be unconditionally fair, but also restricts the actual program 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 assumption about the environment (in other words as an additional assumption in definition 4. In that case, the definition of unconditional 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.

3 Specified Programs

A specification is 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 $s' \models P$.

Predicates will often be characterised by first order formulas. 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, 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 implementation. Given an environment which satisfies the assumptions, then an implementation is required either to *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 implementation 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 $(S(\sigma_1), S(\sigma_j)) \models E$.*

(As in LSP, see [Stø91b], it is also possible to interpret the wait-condition as an assumption about the environment. In that case the environment is assumed always eventually to provide a state which falsifies the wait-condition, while the implementation is required to terminate. The deduction rules are exactly the same for both interpretations.)

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

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

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 implementation. More formally:

Definition 6 $\models_u$ z $\underline{\mathrm{sat}}$ $(\vartheta, \{\})::(P, R, W, G, E)$ *iff* $\mathit{ext}(\vartheta, P, R) \cap \mathit{cp}_u(z) \subseteq \mathit{int}(\vartheta, W, G, E)$.

So far very little has been said about the use of auxiliary variables. 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 the Owicki/Gries method [OG76] (and in many other approaches) auxiliary variables are first implemented as ordinary programming variables and thereafter removed. The reason why this strategy is chosen by Owicki/Gries, is that they conduct their proofs in several iterations, and one way to 'remember' the auxiliary structure from one iteration to the next is to store it in terms of program code. They first conduct a proof in the style of ordinary Hoare-logic, then they prove freedom from interference and so on, and the auxiliary structure is implemented in order to ensure that it remains unchanged from the first to the last iteration. In the method presented in this paper, there is only one proof iteration and therefore no need to 'remember' the auxiliary structure. Thus, the use of auxiliary variables can be 'simulated' in the deduction rules. Note 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.

To characterise validity when the set of auxiliary variables is non-empty, it is necessary to introduce some new notation. 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* with respect to two sets of variables ϑ and α, is the least binary relation $\overset{(\vartheta,\alpha)}{\hookrightarrow}$ on programs such that either:

- $v{:=}\, r \overset{(\vartheta,\alpha)}{\hookrightarrow} v \circ a{:=}\, r \circ u$, where
  - $a \leftarrow_{(\vartheta,\alpha)} u$,
- $\mathsf{blo}\ x_1{:}\, T_1, \ldots, x_n{:}\, T_n\ \mathsf{in}\ z\ \mathsf{olb} \overset{(\vartheta,\alpha)}{\hookrightarrow} \mathsf{blo}\ x_1{:}\, T_1, \ldots, x_n{:}\, T_n\ \mathsf{in}\ z'\ \mathsf{olb}$, where
  - $z \overset{(\vartheta \cup \bigcup_{j=1}^{n} \{x_j\}, \alpha)}{\hookrightarrow} z'$,
- $z_1; z_2 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_1'; z_2'$, where
  - $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_1'$ and $z_2 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2'$,

- if b then z_1 else z_2 fi $\overset{(\vartheta,\alpha)}{\hookrightarrow}$ blo b': B in $b' \circ a := b \circ u$; if b' then z_1' else z_2' fi olb, where
  - $b' \notin \vartheta \cup \alpha$, $a \leftarrow_{(\vartheta,\alpha)} u$, $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_1'$, and $z_2 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2'$,
- while b do z od $\overset{(\vartheta,\alpha)}{\hookrightarrow}$ blo b': B in $b' \circ a := b \circ u$; while b' do z'; $b' \circ a := b \circ u$ od 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 $z_2 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2'$.

The idea is that $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2$ if z_2 can be obtained from z_1 by adding auxiliary structure with respect to a set of programming variables ϑ and a set of auxiliary variables α. 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 the different auxiliary variables do not depend upon each other. The latter requirement makes it possible to remove some auxiliary variables from a specified program without having to remove all the auxiliary variables. This requirement states that if an auxiliary variable occurs on the left-hand side of an assignment-statement, the only auxiliary variable that may occur in the corresponding expression on the right-hand side is the very same variable. However, an assignment to an auxiliary variable may have any number of elements of ϑ in its right-hand side expression.

The block-statement is used in the augmentations of if- and while-statements to allow auxiliary variables to be updated in the same atomic step as the Boolean test is evaluated. Note that the introduced Boolean variable is local and can therefore not be accessed by the environment. Thus the augmentations of if- and while-statements do not significantly change their external behaviour. An internal transition represents either the execution of a declaration list in a block-statement, a Boolean test in an if- or a while-statement, or an assignment-statement. Since the former is 'independent' of the state in which it takes place, it is enough to update auxiliary variables in connection with the execution of Boolean tests and assignment-statements.

Observe that the definition of an augmentation does not restrict the auxiliary variables to be of a particular type. For example, if z_1 denotes the program

```
if $x = 0$ then
      $x{:}= 1$
else
      while $x \leq 0$ do $x{:}= x + 1$ od
fi
```

and z_2 denotes the program

```
blo $b_1{:}$ B in
      $b_1, a{:}= x = 0, a + 1;$
      if $b_1$ then
            $x, a{:}= 1, a + 1$
      else
            blo $b_2{:}$ B in
                  $b_2, a{:}= x \leq 0, a + 1;$
                  while $b_2$ do $x, a{:}= x + 1, a + 1; b_2, a{:}= x \leq 0, a + 1$ od
            olb
      fi
olb
```

then $z_1 \overset{(\{x\},\{a\})}{\hookrightarrow} z_2$.

It may be argued that the definition of an augmentation is rather complicated and hard to remember. However, augmentations are only used to characterise the semantics of a specified program and is not something the user needs to worry about in order to apply the method. For example, augmentations do not occur in the deduction rules.

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 implementation. More formally:

Definition 7 $\models_u z_1 \ \underline{\text{sat}} \ (\vartheta, \alpha){::}(P, R, W, G, E)$ *iff there is a program* z_2*, such that* $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2$ *and* $\models_u z_2 \ \underline{\text{sat}} \ (\vartheta \cup \alpha, \{\ \}){::}(P, R, W, G, E)$.

4 Deduction Rules

The next step is to define a logic, called LSP$_u$, for the deduction of valid specified programs. The consequence-, assignment-, sequential-, while- and parallel-rules are explained in detail below. The remaining rules needed to prove semantic completeness together with some useful adaptation rules are listed in the appendix.

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 $s_1 s_2 \ldots s_k \ldots$ such that for all $j \geq 1$, $(s_j, s_{j+1}) \models C$.

The *consequence-rule*

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

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

The first version of the *assignment-rule*

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

is sufficient whenever the set of auxiliary variables is empty. Any unconditionally 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

$$\begin{array}{ll}
\overleftarrow{P} \wedge R \Rightarrow P & \\
\overleftarrow{P} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \langle v \rangle} \wedge a = \overleftarrow{u} \wedge I_{\alpha \setminus \langle a \rangle} \Rightarrow (G \vee I_{\vartheta \cup \alpha}) \wedge E & \qquad a \leftarrow_{(\vartheta, \alpha)} u \\
\cline{1-1}
v := r \ \underline{\mathrm{sat}} \ (\vartheta, \alpha) :: (P, R, \mathsf{false}, G, R^* \mid E \mid R^*) &
\end{array}$$

is sufficient. The only real difference from above is that the 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. Moreover, since skip is an alias for the assignment of an empty list of expressions to an empty list of variables, the *skip-rule*

$$\begin{array}{l} \overleftharpoon{P} \wedge R \Rightarrow P \\ \overleftharpoon{P} \wedge I_{\vartheta} \wedge a = \overleftharpoon{u} \wedge I_{\alpha \setminus \langle a \rangle} \Rightarrow (G \vee I_{\vartheta \cup \alpha}) \wedge E \\ \hline \mathsf{skip}\ \underline{\mathrm{sat}}\ (\vartheta, \alpha) :: (P, R, \mathsf{false}, G, R^* \mid E \mid R^*) \end{array} \qquad a \leftarrow_{(\vartheta,\alpha)} u$$

follows as a special case.

The *sequential-rule*

$$\begin{array}{l} z_1\ \underline{\mathrm{sat}}\ (\vartheta, \alpha) :: (P_1, R, W, G, P_2 \wedge E_1) \\ z_2\ \underline{\mathrm{sat}}\ (\vartheta, \alpha) :: (P_2, R, W, G, E_2) \\ \hline z_1; z_2\ \underline{\mathrm{sat}}\ (\vartheta, \alpha) :: (P_1, R, W, G, E_1 \mid E_2) \end{array}$$

depends upon the fact that the first component's effect-condition implies the second component's pre-condition. This explains why P_2 occurs in the effect-condition of the first premise. Since an effect-condition covers interference both before the first internal transition and after the last, it follows from the two premises that the overall effect is characterised by $E_1 \mid E_2$.

The *while-rule*

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

can be used when 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 follows 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 well-founded predicate in the second premise. To prove that this premise implies that the statement terminates unless it ends up busy-waiting in W, assume there is a non-terminating computation (a computation where the empty program is never reached)

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

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 busy-waiting in W.

In the general case the following rule

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

is needed. Remember that the execution of while b do z od corresponds to the execution of a program of the form

$$\mathsf{blo}\ b' \colon \mathsf{B}\ \mathsf{in}\ b' \circ a := b \circ u; \mathsf{while}\ b'\ \mathsf{do}\ z'; b' \circ a := b \circ u\ \mathsf{od}\ \mathsf{olb},$$

where $a \leftarrow_{(\vartheta,\alpha)} u$ and $z \overset{(\vartheta,\alpha)}{\hookrightarrow} z'$. The third premise simulates the evaluation of the Boolean test. Thus, the overall effect satisfies $R^* \mid (E_1 \mid E_2)^+ \mid (E_1 \wedge \neg b) \mid R^*$ if the loop iterates at least once, and $R^* \mid (E_1 \wedge \neg b) \mid R^*$ otherwise.

To grasp the intuition behind the *parallel-rule*, consider first the rule

$$\begin{array}{l}
z_1\ \underline{\mathrm{sat}}\ (\vartheta, \alpha) :: (P, R \vee G_2, \mathsf{false}, G_1, E_1) \\
z_2\ \underline{\mathrm{sat}}\ (\vartheta, \alpha) :: (P, R \vee G_1, \mathsf{false}, G_2, E_2) \\
\hline
\{z_1 \parallel z_2\}\ \underline{\mathrm{sat}}\ (\vartheta, \alpha) :: (P, R, \mathsf{false}, G_1 \vee G_2, E_1 \wedge E_2)
\end{array}$$

which is sufficient whenever both component programs terminate. Observe that the rely-condition of the first premise allows any interference due to z_2 (given the actual assumptions about the overall environment), and similarly that the rely-condition of the second 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 the two premises that $\{z_1 \parallel z_2\}$ terminates, that any internal transition, which changes the global state, satisfies $G_1 \vee G_2$, and that the overall effect satisfies $E_1 \wedge E_2$.

The next version of the parallel-rule

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

is sufficient whenever the overall program $\{z_1 \parallel z_2\}$ terminates. It follows from the second premise that z_1 can end up busy-waiting only in 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 busy-waiting only in 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 busy-waiting after z_2 has terminated, that z_2 cannot be busy-waiting after z_1 has terminated, and that z_1 and z_2 cannot be busy waiting at the same time, it follows that $\{z_1 \parallel z_2\}$ is guaranteed to terminate in an environment characterised by P and R.

It is now easy to extend the rule to deal with the general case:

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

The idea is that W characterises the states in which the overall program is allowed to end up busy-waiting.

This rule can of course be generalised further to deal with more than two processes:

$$\frac{\begin{array}{l}\neg(W_j \wedge \bigwedge_{k=1,k\neq j}^{m}(W_k \vee E_k))_{1\leq j\leq m}\\ z_j \ \underline{\mathrm{sat}}\ (\vartheta,\alpha)::(P, R \vee \bigvee_{k=1,k\neq j}^{m} G_k, W \vee W_j, G_j, E_j)_{1\leq j\leq m}\end{array}}{\parallel_{j=1}^{m} z_j \ \underline{\mathrm{sat}}\ (\vartheta,\alpha)::(P, R, W, \bigvee_{j=1}^{m} G_j, \bigwedge_{j=1}^{m} E_j)}$$

Here, $\parallel_{j=1}^{m} z_j$ denotes any program that can be obtained from $z_1 \parallel \ldots \parallel z_m$ by adding curly brackets. The 'first' premise ensures that whenever process j busy-waits in a state s such that $s \models \neg W \wedge W_j$, then there is at least one other process which has not terminated and is not busy-waiting. This rule is 'deducible' from the basic rules of LSP$_u$.

5 Examples

Examples where a logic of this type is employed for the development of non-trivial programs can be found in [Stø90], [Stø91a], [XH92]. Moreover, it is shown in [Stø90], [Stø91b] how auxiliary variables can be used both as a specification tool to eliminate undesirable implementations, and as a verification tool to make it possible to prove that an already finished program satisfies a certain specification. The object here is to apply LSP$_u$ to prove fair termination.

Let z_1 and z_2 denote the programs

$$b := \mathsf{true}, \qquad\qquad \mathsf{while}\ \neg b\ \mathsf{do\ skip\ od}.$$

It should be obvious that

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

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

$$\vdash_u z_1\ \underline{\mathrm{sat}}\ (\{b\},\{\,\})::(\mathsf{true}, \overleftarrow{b} \Rightarrow b, \mathsf{false}, \overleftarrow{b} \Rightarrow b, b),$$
$$\vdash_u z_2\ \underline{\mathrm{sat}}\ (\{b\},\{\,\})::(\mathsf{true}, \overleftarrow{b} \Rightarrow b, \neg b, \overleftarrow{b} \Rightarrow b, \mathsf{true}).$$

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

$$\vdash_u \mathsf{skip}\ \underline{\mathrm{sat}}\ (\{b\},\{\,\})::(\neg b, \overleftarrow{b} \Rightarrow b, \neg b, \overleftarrow{b} \Rightarrow b, \mathsf{true}),$$

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.

A slightly more complicated synchronisation is dealt with in the next example. Let z_1 and z_2 denote the programs

```
while $n > 0$ do
   if $n \bmod 2 = 0$ then $n := n$-1 else skip fi
od,

while $n > 0$ do
   if $n \bmod 2 = 1$ then $n := n$-1 else skip fi
od.
```

Given that $n > 0$, the program z_1 may reduce the value of n by 1 if n is even, while z_2 may subtract 1 from n if n is odd. Thus, it should be clear that

$$\models_u \{z_1 \parallel z_2\}\ \underline{\mathrm{sat}}\ (\{n\},\{\,\})::(\mathsf{true}, n < \overleftarrow{n}, \mathsf{false}, \mathsf{true}, n \le 0).$$

Moreover, this follows by the consequence- and parallel-rules if

$$\vdash_u z_1\ \underline{\mathrm{sat}}\ (\{n\},\{\,\})::(\mathsf{true}, n < \overleftarrow{n}, n \bmod 2 = 1 \wedge n > 0, n < \overleftarrow{n}, n \le 0),$$
$$\vdash_u z_2\ \underline{\mathrm{sat}}\ (\{n\},\{\,\})::(\mathsf{true}, n < \overleftarrow{n}, n \bmod 2 = 0 \wedge n > 0, n < \overleftarrow{n}, n \le 0).$$

The first of these can be deduced by the pre-, consequence-, assignment-, if- and while-rules since it can easily be proved that z_1's if-statement satisfies

$$(\{n\},\{\})::(n>0, n<\overleftarrow{n},$$
$$n \bmod 2 = 1 \wedge n>0, n<\overleftarrow{n}, \overleftarrow{n} \bmod 2 = 0 \Rightarrow n<\overleftarrow{n})$$

and

$$\overleftarrow{n}>0 \wedge (\overleftarrow{n} \bmod 2 = 0 \Rightarrow n<\overleftarrow{n}) \wedge$$
$$(n \le \overleftarrow{n}) \mid (n = \overleftarrow{n} \wedge (n \bmod 2 = 0 \vee n \le 0)) \mid (n \le \overleftarrow{n})$$

is well-founded. Not surprisingly, z_2 can be proved to satisfy its specification in a similar way.

Finally, to indicate how LSP$_u$ can be used for the design of programs in a top-down style, let g be a function, such that $\models \exists y \cdot \in Z \colon g(y) = 0$, and consider the task of designing a program z which satisfies

$$\vdash_u z \ \underline{\text{sat}}\ (\{x\},\{\})::(\mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{true}, g(x)=0).$$

One sensible decomposition strategy is two split the searching into two parallel processes z_1 and z_2 dealing with respectively the non-negative and the negative integers. This means that z should be of the form

$$\mathsf{blo}\ f\colon \mathsf{B}\ \mathsf{in}\ f := \mathsf{false}; \{z_1 \parallel z_2\}\ \mathsf{olb},$$

where the Boolean flag f is to be switched on when the appropriate argument is found. Clearly, any atomic step after f has been initialised is required to satisfy the *binary* invariant

$$(\overleftarrow{f} \Rightarrow f) \wedge (f \Rightarrow g(x) = 0),$$

from now on denoted by in. Moreover, it follows by the effect-, consequence-, assignment-, sequential- and block-rules that this is a valid implementation of z if

$$\vdash_u \{z_1 \parallel z_2\}\ \underline{\text{sat}}\ (\{x,f\},\{\})::(\neg f, in, \mathsf{false}, in, f),$$

which again can be deduced by the consequence- and parallel-rules if

$$\vdash z_1\ \underline{\text{sat}}\ (\{x,f\},\{\})::(\neg f, in, (\forall y \in Z \colon y \ge 0 \Rightarrow g(y) \ne 0) \wedge \neg f, in, f),$$
$$\vdash z_2\ \underline{\text{sat}}\ (\{x,f\},\{\})::(\neg f, in, (\forall y \in Z \colon y < 0 \Rightarrow g(y) \ne 0) \wedge \neg f, in, f).$$

Moreover, the consequence-, assignment-, sequential- and block-rules imply that

$$\mathsf{blo}\ x' \colon Z\ \mathsf{in}\ x' := 0; z_1'\ \mathsf{olb}$$

is a valid decomposition of z_1 if

$$z_1'\ \underline{\text{sat}}\ (\{x,f,x'\},\{\})::(\neg f \wedge x' = 0, in \wedge x' = x',$$
$$(\forall y \in Z \colon y \ge 0 \Rightarrow g(y) \ne 0) \wedge \neg f, in, f).$$

Finally, since

$$\begin{aligned}&(\neg \overleftharpoon{f} \wedge \overleftharpoon{x'} \geq 0 \wedge (\forall y \in Z{:}\, 0 \leq y < \overleftharpoon{x'} \Rightarrow f(y) \neq 0) \wedge \\ &\qquad (g(\overleftharpoon{x'}) = 0 \Rightarrow f) \wedge (g(\overleftharpoon{x'}) \neq 0 \Rightarrow x' = \overleftharpoon{x'} + 1))^{+} \wedge \\ &in \mid (x = \overleftharpoon{x} \wedge (f \Leftrightarrow \overleftharpoon{f}) \wedge x' = \overleftharpoon{x'} \wedge (\exists y \cdot \in Z{:}\, y \geq 0 \wedge g(y) = 0 \vee f)) \mid in\end{aligned}$$

is well-founded, it can be deduced by the pre-, consequence-, assignment-, if- and while-rules that

$$\mathsf{while}\ \neg f\ \mathsf{do}\ \mathsf{if}\ g(x') = 0\ \mathsf{then}\ f, x{:=}\ \mathsf{true}, x'\ \mathsf{else}\ x'{:=}\ x' + 1\ \mathsf{fi}\ \mathsf{od}$$

is a correct implementation of z_1'. A program which implements z_2 can be designed in a similar style.

6 Discussion

Rules for proving fair termination with respect to a set of transition functions $\{f_1, \ldots, f_m\}$ assosiated with m processes are described in [LPS81]. A state predicate is used to characterise helpful directions in the same way as LSP$_u$ employs a wait-condition to select helpful execution paths of the while-statement's body. Rules for fair termination with respect to a set of transition functions are also given in [APS84]. These rules are based upon explicit scheduling.

Explicit scheduling is also used in [OA86], but in a less abstract setting. Unfortunately, the method 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 large software products 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 proof method should satisfy what is known as the principle of *compositionality* [dR85], [Zwi89] — namely that a program's specification always can be verified on the basis of the specifications of its constituent components, without knowledge of the interior program structure of those components.

The methods in [BKP84] and [Lam85] are based upon temporal logic. They are both compositional and non-transformational. Moreover, they can be used to prove total correctness with respect to the type of programming language discussed above. However, due to lack of published examples, it is not clear how useful they are when it comes to practical program development. One obvious difference, with respect to the approach presented in this paper, is that these logics have a much wider application area — they can for example be used for the design of non-terminating programs with respect to general liveness properties. Some very general compositional parallel-rules are proposed in [Sta85] and [AL90].

LSP is a compositional formal method specially designed for top-down development of totally correct shared-state parallel programs. LSP can be thought of as a compositional reformulation of the Owicki/Gries method [OG76], and also as an extension of Jones' rely/guarantee approach [Jon83]. A related system is described in [XH91]. Examples where LSP is used for the development of non-trivial programs can be found in [Stø90], [Stø91a]. In [Stø90] it is also explained how LSP can be extended to deal with partial functions and guarded commands.

This paper shows how LSP can be modified to allow for the design of programs, whose correctness depends upon busy-waiting. Only unconditional fairness is discussed here. However, systems for weak and strong fairness can be formulated in a similar style. The author is currently working on a paper which deals with both unconditional, weak and strong fairness. This paper will include soundness and semantic completeness proofs for the formal system presented above.

7 Acknowledgements

I would like to thank my PhD supervisor Cliff B. Jones for his help and support. I am also indebted to Howard Barringer and Xu Qiwen. A special thanks goes to Mathai Joseph whose comments led to many improvements.

The research reported in this paper was carried out at the Department of Computer Science, Manchester University with financial support from the Norwegian Research Council for Science and the Humanities and the Wolfson Foundation.

References

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

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

[APS84] K. R. Apt, A. Pnueli, and J. Stavi. Fair termination revisited with delay. *Theoretical Computer Science*, 33:65–84, 1984.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proc. Sixteenth ACM Symposium on Theory of Computing*, pages 51–63, 1984.

[dR85] W. P. de Roever. The quest for compositionality, formal models in programming. In F.J. Neuhold and C. Chroust, editors, *Proc. IFIP 85*, pages 181–205, 1985.

[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.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th ACM-POPL*, 1980.

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

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

[Lam85] L. Lamport. An axiomatic semantics of concurrent programming languages. In K. R. Apt, editor, *Logics and Models of Concurrent Systems.* NATO ASI Series, Vol. F13, 1985.

[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.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[Par81] D. Park. A predicate transformer for weak fair iteration. In *Proc. 6th IBM Symp. on Math. Foundation of Computer Science*, 1981.

[Sta85] E. W. Stark. A proof technique for rely/guarantee properties. In S.N. Maheshwari, editor, *Proc. 5th Conference on the Foundation of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 206*, pages 369–391, 1985.

[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. Also available as technical report UMCS-91-7-1, University of Manchester.

[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. Also available as technical report UMCS-91-6-1, University of Manchester.

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

[XH92] Q. Xu and J. He. A case study in formally developing state-based parallel programs — the dutch national torus. In C.B. Jones and R.C. Shaw, editors, *Proc. 5th BCS-FACS Refinement Workshop*. 1992.

[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-Verlag 1989.

Additional Rules Needed to Prove Semantic Completeness

if: :

$$\begin{array}{l}
\overleftarrow{P_1} \wedge R \Rightarrow P_1 \\
\mathsf{skip}\ \underline{\mathrm{sat}}\ (\vartheta, \alpha)::(P_1, \mathsf{false}, \mathsf{false}, G, P_2 \wedge E_1) \\
z_1\ \underline{\mathrm{sat}}\ (\vartheta, \alpha)::(P_2 \wedge b, R, W, G, E_2) \\
z_2\ \underline{\mathrm{sat}}\ (\vartheta, \alpha)::(P_2 \wedge \neg b, R, W, G, E_2) \\
\hline
\mathsf{if}\ b\ \mathsf{then}\ z_1\ \mathsf{else}\ z_2\ \mathsf{fi}\ \underline{\mathrm{sat}}\ (\vartheta, \alpha)::(P_1, R, W, G, R^* \mid E_1 \mid E_2)
\end{array}$$

block: :

$$\begin{array}{l}
z\ \underline{\mathrm{sat}}\ (\vartheta, \alpha)::(P, R \wedge \bigwedge_{j=1}^{n} x_j = \overleftarrow{x_j}, W, G, E) \\
\hline
\mathsf{blo}\ x_1 \colon T_1, \ldots, x_n \colon T_n\ \mathsf{in}\ z\ \mathsf{olb}\ \underline{\mathrm{sat}}\ (\vartheta \setminus \bigcup_{j=1}^{n}\{x_j\}, \alpha)::(P, R, W, G, E)
\end{array}$$

elimination: :

$$\begin{array}{l}
x \notin \vartheta \\
z\ \underline{\mathrm{sat}}\ (\vartheta, \alpha)::(P, R, W, G, E) \\
\hline
z\ \underline{\mathrm{sat}}\ (\vartheta, \alpha \setminus \{x\})::(\exists x \cdot\colon P, \forall \overleftarrow{x} \colon \exists x \cdot\colon R, W, G, E)
\end{array}$$

pre: :

$$\begin{array}{l}
z\ \underline{\mathrm{sat}}\ (\vartheta, \alpha)::(P, R, W, G, E) \\
\hline
z\ \underline{\mathrm{sat}}\ (\vartheta, \alpha)::(P, R, W, G, \overleftarrow{P} \wedge E)
\end{array}$$

Some Useful Adaptation Rules

effect: :

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

rely: :

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

invariant: :

$$\frac{\begin{array}{l} P \Rightarrow K \\ \overleftarrow{K} \wedge (R \vee G) \Rightarrow K \\ z\ \underline{\text{sat}}\ (\vartheta,\alpha)::(P,R,W,G,E) \end{array}}{z\ \underline{\text{sat}}\ (\vartheta,\alpha)::(P,R,K \wedge W,\overleftarrow{K} \wedge G,E)}$$

stutter: :

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

glo: :

$$\frac{\begin{array}{l} x \notin \vartheta \cup \alpha \\ z\ \underline{\text{sat}}\ (\vartheta,\alpha)::(P,R,W,G,E) \end{array}}{z\ \underline{\text{sat}}\ (\vartheta \cup \{x\},\alpha)::(P,R,W,G \wedge x = \overleftarrow{x},E)}$$

aux: :

$$\frac{\begin{array}{l} x \notin \vartheta \cup \alpha \\ z\ \underline{\text{sat}}\ (\vartheta,\alpha)::(P,R,W,G,E) \end{array}}{z\ \underline{\text{sat}}\ (\vartheta,\alpha \cup \{x\})::(P,R,W,G \wedge x = \overleftarrow{x},E)}$$