A Method for the Development of Totally Correct Shared-State Parallel Programs

Ketil Stølen,
Department of Computer Science, Manchester University,
Oxford Road, Manchester, M13, 9PL
email:ks@uk.ac.man.cs

Abstract

A syntax-directed formal system for the development of totally correct programs with respect to an (unfair) shared-state parallel programming language is proposed. The programming language is basically a while-language extended with parallel- and await-constructs. The system is called LSP (Logic of Specified Programs) and can be seen of as an extension of Jones' rely/guarantee method. His method is strengthened in two respects:

- Specifications are extended with a wait-condition to allow for the development of programs whose correctness depends upon synchronisation. The wait-condition is supposed to characterise the states in which the implementation may become blocked. The implementation is not allowed to become blocked inside the body of an await-statement.
- Auxiliary variables are introduced to increase expressiveness. They are either used as a specification tool to eliminate undesirable implementations or as a verification tool to prove that a certain program satisfies a particular specification. Although it is possible to define history variables in LSP, the auxiliary variables may be of any type, and it is up to the user to define the auxiliary structure he prefers. Moreover, the auxiliary structure is only a part of the logic. This means that auxiliary variables do not have to be implemented as if they were ordinary programming variables.

1 Introduction

The parallel-rule in the Owicki/Gries method [OG76] depends upon a number of tests which only can be carried out after the component processes have been implemented and their proofs have been constructed. For large software products this strategy is unacceptable, because erroneous design decisions, taken early in the design process, may remain undetected until after the whole program is complete. Since, in the worst case, everything that depends upon such mistakes will have to be thrown away, much work could be wasted.

The usual way of avoiding this problem is to specify processes in terms of *assumptions* and *commitments*. This was first proposed by Francez and Pnueli [FP78]. The basic idea is: If the *environment*, by which is meant the set of processes running in parallel with the one in question, fulfills the assumptions, then the actual process is required to fulfill the commitments.

Jones employs rely- and guar(antee)-conditions [Jon83] in a similar style. However, while earlier approaches focus essentially on program verification; the object of the rely/guarantee method is *top-down* program development. The proof tuples are of the form z sat (P, R, G, Q), where z is a program and (P, R, G, Q) is a specification consisting of four assertions P, R, G and Q. The pre-condition P and the rely-condition R constitute assumptions that the developer can make about the environment. In return the implementation z must satisfy the guar-condition G, the post-condition Q, and terminate when operated in an environment which fulfills the assumptions.

The pre-condition characterises a set of states to which the implementation is applicable. Any uninterrupted state transition by the environment is supposed to satisfy the rely-condition, while any atomic state transition by the implementation must satisfy the guar-condition. Finally, the post-condition characterises the overall effect of using the implementation in such an environment.

The rely/guarantee method allows erroneous interference decisions to be spotted and corrected at the level where they are taken. Moreover, specifications are decomposed into subspecifications. Thus programs can be developed in a top-down style. Unfortunately, this approach is inferior to the one proposed by Owicki and Gries in two respects: First of all, the method cannot deal with synchronisation. Hence, programs whose correctness depends upon some sort of delay-construct cannot be developed. Secondly, many valid developments are excluded because sufficiently strong assertions cannot be expressed.

This paper, based on the author's PhD-thesis [Stø90], presents an extension of the rely/guarantee method, called LSP (Logic of Specified Programs), which does not suffer from the two weaknesses pointed out above.

The paper is organised as follows: The next section, section 2, defines the programming language. Section 3 explains how rely- and guar-conditions can be used to deal with interfering programs, while section 4 extends specifications with a wait-condition to facilitate development of synchronised programs. The use of auxiliary variables is covered by section 5. Finally, section 6 indicates some extensions and compares LSP with methods known from the literature.

2 Programming Language

2.1 Syntax

A *program* is a finite, nonempty list of symbols whose context-independent syntax can be characterised in the well-known BNF-notation: Given that $\langle vl \rangle$, $\langle el \rangle$, $\langle dl \rangle$, $\langle ts \rangle$ denote respectively a nonempty list of variables, a nonempty list of expressions, a nonempty list of variable declarations, and a Boolean test, then any program is of the form $\langle pg \rangle$, where

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

The main structure of a program is characterised above. However, a syntactically correct program is also required to satisfy some supplementary constraints:

- Not surprisingly, the assignment-statement's two lists are required to have the same number of elements. Moreover, the j'th variable in the first list must be of the same type as the j'th expression in the second, and the same variable is not allowed to occur in the variable list more than once.

- The block-statement allows for declaration of variables. A variable is *local* to a program, if it is declared in the program; otherwise it is said to be *global*. For example, blo $x{:}N, y{:}N; x, y := 5{+}w, w$ olb has two local variables, x and y, and one global variable w. To avoid complications due to name clashes, it is required that the same variable cannot be declared more than once in the same program, and that a local variable cannot appear outside its block. The first constraint avoids name clashes between local variables, while the second ensures that the set of global variables is disjoint from the set of local variables.

- To simplify the deduction rules and the reasoning with auxiliary variables, it is required that variables occurring in the Boolean test of an if- or a while-statement cannot be accessed by any process running in parallel. This requirement does of course not reduce the number of implementable algorithms. If x is a variable that can be accessed by another process running in parallel, then it is for example always possible to write blo $y{:}N; y := x$; if $y = 0$ then z_1 else z_2 fi olb instead of if $x = 0$ then z_1 else z_2 fi. (The constraint can be removed [Stø90], but the resulting system is more complicated and less intuitive. Similar requirements are stated in [Sou84], [XH91].)

2.2 Operational Semantics

The programming language is given operational semantics in the style of [Acz83]. A *state* is a mapping of all programming variables to values, and a *configuration* is a pair $\langle z, s\rangle$ where z is either a program or the *empty program* ε and s is a state. An *external* transition $\xrightarrow{e}$ 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 $\xrightarrow{i}$ is the least binary relation on configurations such that either:

- $\langle \text{skip}, s\rangle \xrightarrow{i} \langle \varepsilon, s\rangle$,
- $\langle v := r, s\rangle \xrightarrow{i} \langle \varepsilon, s(\substack{v\\r})\rangle$, where $s(\substack{v\\r})$ denotes the state that is obtained from s, by mapping the variables in v to the values of r in the state s, and leaving all other maplets unchanged,
- $\langle \text{blo } d; z \text{ olb}, s_1\rangle \xrightarrow{i} \langle z, s_2\rangle$, 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 $\langle z_1, s_1\rangle \xrightarrow{i} \langle \varepsilon, s_2\rangle$,
- $\langle z_1; z_2, s_1\rangle \xrightarrow{i} \langle z_3; z_2, s_2\rangle$ if $\langle z_1, s_1\rangle \xrightarrow{i} \langle z_3, s_2\rangle$ and $z_3 \neq \varepsilon$,
- $\langle \text{if } b \text{ then } z_1 \text{ else } z_2 \text{ fi}, s\rangle \xrightarrow{i} \langle z_1, s\rangle$ if $s \models b$,
- $\langle \text{if } b \text{ then } z_1 \text{ else } z_2 \text{ fi}, s\rangle \xrightarrow{i} \langle z_2, s\rangle$ if $s \models \neg b$,
- $\langle \text{while } b \text{ do } z \text{ od}, s\rangle \xrightarrow{i} \langle z; \text{while } b \text{ do } z \text{ od}, s\rangle$ if $s \models b$,
- $\langle \text{while } b \text{ do } z \text{ od}, s\rangle \xrightarrow{i} \langle \varepsilon, s\rangle$ if $s \models \neg b$,
- $\langle \{z_1 \parallel z_2\}, s_1\rangle \xrightarrow{i} \langle z_2, s_2\rangle$ if $\langle z_1, s_1\rangle \xrightarrow{i} \langle \varepsilon, s_2\rangle$,
- $\langle \{z_1 \parallel z_2\}, s_1\rangle \xrightarrow{i} \langle z_1, s_2\rangle$ if $\langle z_2, s_1\rangle \xrightarrow{i} \langle \varepsilon, s_2\rangle$,
- $\langle \{z_1 \parallel z_2\}, s_1\rangle \xrightarrow{i} \langle \{z_3 \parallel z_2\}, s_2\rangle$ if $\langle z_1, s_1\rangle \xrightarrow{i} \langle z_3, s_2\rangle$ and $z_3 \neq \varepsilon$,
- $\langle \{z_1 \parallel z_2\}, s_1\rangle \xrightarrow{i} \langle \{z_1 \parallel z_3\}, s_2\rangle$ if $\langle z_2, s_1\rangle \xrightarrow{i} \langle z_3, s_2\rangle$ and $z_3 \neq \varepsilon$,
- $\langle \text{await } b \text{ do } z_1 \text{ od}, s_1\rangle \xrightarrow{i} \langle \varepsilon, s_n\rangle$ if $s_1 \models b$, and
  - there is a list of configurations $\langle z_2, s_2\rangle, \ldots, \langle z_{n-1}, s_{n-1}\rangle$, such that $\langle z_{n-1}, s_{n-1}\rangle \xrightarrow{i} \langle \varepsilon, s_n\rangle$ and for all $1 < k < n$, $\langle z_{k-1}, s_{k-1}\rangle \xrightarrow{i} \langle z_k, s_k\rangle$,
- $\langle \text{await } b \text{ do } z_1 \text{ od}, s_1\rangle \xrightarrow{i} \langle \text{await } b \text{ do } z_1 \text{ od}, s_1\rangle$ if $s_1 \models b$, and

- there is an infinite list of configurations $\langle z_2, s_2\rangle, \ldots, \langle z_n, s_n\rangle, \ldots$, such that for all $k > 1$, $\langle z_{k-1}, s_{k-1}\rangle \xrightarrow{i} \langle z_k, s_k\rangle$, or
- there is a finite list of configurations $\langle z_2, s_2\rangle, \ldots, \langle z_n, s_n\rangle$, such that $z_n \neq \varepsilon$ and there is no configuration $\langle z_{n+1}, s_{n+1}\rangle$ which satisfies $\langle z_n, s_n\rangle \xrightarrow{i} \langle z_{n+1}, s_{n+1}\rangle$, and for all $1 < k \leq n$, $\langle z_{k-1}, s_{k-1}\rangle \xrightarrow{i} \langle z_k, s_k\rangle$.

The above definition is of course only sensible if all functions are required to be total. It follows from the definition that assignment-statements and Boolean tests are atomic. The environment is restricted from interfering until an await-statement's body has terminated if an evaluation of its Boolean test comes out true. Moreover, the execution of the await-statement's body is modeled by one atomic step. The identity internal transition is used to model that the await-statement's body fails to terminate.

Before giving the definition of a computation, there are two things which must be sorted out: First of all, since the object here is to prove total correctness, a progress property is needed. Secondly, since the environment is restricted from accessing certain variables, it is necessary to find a way to constrain them from being updated by external transitions.

Nobody doubts that the sequential program $x := 1$ (given its usual semantics) eventually will terminate. The reason is that any sensible sequential programming language satisfies the following progress property: If something can happen then eventually something will happen.

In the concurrent case, with respect to an unfair programming language, a slightly different progress property is required. Given that a configuration c_1 is *disabled* if there is no configuration c_2, such that $c_1 \xrightarrow{i} c_2$, then it is enough to insist that the final configuration of a finite computation is disabled. This constrains programs from not progressing without being disabled or infinitely overtaken by the environment. Observe that this is no fairness constraint, because it does not restrict the actual program from being infinitely overtaken by the environment.

To deal with the environment's restricted access to certain variables, let for any program z, $hid[z]$ denote the set of any variable which is local to z or occurs in the Boolean test of an if- or a while-statement in z — in which case it is sufficient to constrain the external transitions in a computation of z from changing the values of the variables in $hid[z]$.

Definition 1 *A* computation *is a possibly infinite sequence of external and internal transitions*

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

such that the final configuration is disabled if the sequence is finite, and the external transitions leave the values of the variables in $hid[z_1]$ unchanged.

Given a computation σ, then $Z(\sigma)$, $S(\sigma)$ and $L(\sigma)$ are the obvious projection functions to sequences of programs, states and transition labels, while $Z(\sigma_j)$, $S(\sigma_j)$, $L(\sigma_j)$ and σ_j denote respectively the j'th program, the j'th state, the j'th transition label and the j'th configuration. Furthermore, $\sigma(j, \ldots, k)$ represents

$$\langle Z(\sigma_j), S(\sigma_j)\rangle \xrightarrow{L(\sigma_j)} \quad \ldots \quad \xrightarrow{L(\sigma_{k-1})} \langle Z(\sigma_k), S(\sigma_k)\rangle.$$

If σ is infinite, then $len(\sigma) = \infty$, otherwise $len(\sigma)$ is equal to σ's number of configurations. Finally, given a set of variables ϑ and two states s_1, s_2, then $s_1 \overset{\vartheta}{=} s_2$ denotes that for all variables $x \in \vartheta$, $s_1(x) = s_2(x)$, while $s_1 \overset{\vartheta}{\neq} s_2$ means that there is a variable $x \in \vartheta$, such that $s_1(x) \neq s_2(x)$.

Definition 2 *Given a program z, let $cp[z]$ be the set of all computations σ such that $Z(\sigma_1) = z$.*

The computations σ' of z_1 and σ'' of z_2 are *compatible* if $\{z_1 \parallel z_2\}$ is a program, $len(\sigma') = len(\sigma'')$, $S(\sigma') = S(\sigma'')$ and for all $1 \leq j < len(\sigma')$, $L(\sigma'_j) = L(\sigma''_j)$ implies $L(\sigma'_j) = e$. For example, given three assignment-statements z_1, z_2 and z_3, then

$$\langle z_1; z_2, s_1\rangle \xrightarrow{i} \langle z_2, s_2\rangle \xrightarrow{i} \langle \varepsilon, s_3\rangle$$

is a computation of $z_1; z_2$ which is not compatible with any computation of z_3, because each finite computation of z_3 has exactly one internal transition and the computation above has no external transitions. On the other hand, the computations

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

are compatible. Furthermore, they can be composed into a computation

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

of $\{z_1; z_2 \parallel z_3\}$, by composing the program part of each configuration pair, and making a transition internal iff one of the two component transitions are internal. It can easily be shown that two compatible computations σ' of z_1 and σ'' of z_2 can always be composed into a computation of $\{z_1 \parallel z_2\}$. Furthermore, it is also straightforward to prove that any computation σ of a parallel program $\{z_1 \parallel z_2\}$ can be decomposed into two compatible computations σ' of z_1 and σ'' of z_2, such that σ is the result of composing them.

3 Interference

3.1 Specified Programs

The object of this section is to show how rely- and guar-conditions can be used for the development of interfering programs. The formalism presented below can be seen as a slightly altered version of Jones' rely/guarantee method. The main modifications are that the reflexivity and transitivity constraints on rely- and guar-conditions have been removed. Moreover, the post-condition is called eff-condition, and the proof-tuples have an extra component characterising the set of global programming variables.

The base logic L is a μ-calculus. In the style of VDM [Jon90] hooked variables will be used to refer to an earlier state (which is not necessarily the previous state). This means that, for any *unhooked* variable x of type Σ, there is a *hooked* variable $\overleftarrow{x}$ of type Σ. Hooked variables are restricted from occurring in programs.

Given a structure and a valuation then the expressions in L can be assigned meanings in the usual way; $\models A$ means that the assertion A is valid (in the actual structure), while $(s_1, s_2) \models A$, where (s_1, s_2) is a pair of states, means that A is true if each hooked variable x in A is assigned the value $s_1(x)$ and each unhooked variable x in A is assigned the value $s_2(x)$. The first state s_1 may be omitted if A has no occurrences of hooked variables.

Thus, an assertion A can be interpreted as the set of all pairs of states (s_1, s_2), such that $(s_1, s_2) \models A$. If A has no occurrences of hooked variables, it may also be thought of as the set of all states s, such that $s \models A$. Both interpretations will be used below. To indicate the intended meaning, it will be distinguished between *binary* and *unary* assertions. When an assertion is binary it denotes a set of pairs of states, while an unary assertion denotes a set of states. In other words, an assertion with occurrences of hooked variables is always binary, while an assertion without occurrences of hooked variables can be both binary and unary.

A specification is of the form $(\vartheta)::(P, R, G, E)$, where the *pre-condition* P is a unary assertion, the *rely-condition* R, the *guar-condition* G, and the *eff-condition* E are binary assertions. The *glo-set* ϑ is the set of global programming variables. It is required that the unhooked version of any hooked or unhooked free variable occurring in P, R, G or E is an element of ϑ. The *global state* is the state restricted to the glo-set.

A specification states a number of assumptions about the environment. First of all, the initial state is assumed to satisfy the pre-condition. Moreover, it is also assumed that any external transition which

changes the global state satisfies the rely-condition. For example, given the rely-condition $x < \overleftarrow{x} \wedge y = \overleftarrow{y}$, then it is assumed that the environment will never change the value of y. Furthermore, if the environment assigns a new value to x, then this value will be less than or equal to the variable's previous value.

Thirdly, it is assumed that the environment can only perform a finite number of consecutive atomic steps. This means that no computation has an infinite number of external transitions unless it also has an infinite number of internal transitions. Thus, this assumption implies that the implementation will not be *infinitely overtaken* by the environment. Observe that this is not a fairness requirement on the programming language, because it does not constrain the implementation of a specification. If for example a parallel-statement $\{z_1 \parallel z_2\}$ occurs in the implementation, then this assumption does not influence whether or not z_1 is infinitely overtaken by z_2. Moreover, this assumption can be removed. The only difference is that an implementation is no longer required to *terminate*, but only to terminate whenever it is not infinitely overtaken by the environment (see [Stø90] for a more detailed discussion). The assumptions are summed up in the definition below:

Definition 3 *Given a glo-set ϑ, a pre-condition P, a rely-condition R, then $ext[\vartheta, P, R]$ denotes the set of all computations σ, such that:*

- $S(\sigma_1) \models P$,
- *for all* $1 \leq j < len(\sigma)$, *if* $L(\sigma_j) = e$ *and* $S(\sigma_j) \overset{\vartheta}{\neq} S(\sigma_{j+1})$ *then* $(S(\sigma_j), S(\sigma_{j+1})) \models R$,
- *if* $len(\sigma) = \infty$, *then for all* $j \geq 1$, *there is a* $k \geq j$, *such that* $L(\sigma_k) = i$.

A specification is of course 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 to terminate. Moreover, any internal transition which changes the global state is required to satisfy the guar-condition, and the overall effect is constrained to satisfy the eff-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 eff-condition for the program skip is $x \geq \overleftarrow{x}$. The commitments are summed up below:

Definition 4 *Given a glo-set ϑ, a guar-condition G, and an eff-condition E, then $int[\vartheta, G, E]$ denotes the set of all computations σ, such that:*

- $len(\sigma) \neq \infty$ *and* $Z(\sigma_{len(\sigma)}) = \varepsilon$,
- *for all* $1 \leq j < len(\sigma)$, *if* $L(\sigma_j) = i$ *and* $S(\sigma_j) \overset{\vartheta}{\neq} S(\sigma_{j+1})$ *then* $(S(\sigma_j), S(\sigma_{j+1})) \models G$,
- $(S(\sigma_1), S(\sigma_{len(\sigma)})) \models E$.

A specified program is a pair of a program z and a specification ψ, written z sat ψ. It is required that for any variable x occurring in z, x is an element of ψ's glo-set iff x is a global variable with respect to z. Moreover, a specified program is valid if the program behaves according to the commitments whenever it is executed in an environment which satisfies the assumptions:

Definition 5 *A specified program z sat $(\vartheta)::(P, R, G, E)$ is valid iff $ext[\vartheta, P, R] \cap cp[z] \subseteq int[\vartheta, G, E]$.*

3.2 Deduction Rules

The object of this section is to define a logic for the deduction of valid specified programs. Given a list of variables v, a list of expressions r, a finite set of variables ϑ, and three assertions A, B and C, where at least A is unary, then $\langle v \rangle$ denotes the set of variables occurring in v, $\overleftarrow{r}$ denotes the list of expressions that can be obtained from r by hooking all free variables in r, $\overleftarrow{A}$ denotes the assertion that can be obtained from A by hooking all free variables in A, I_ϑ denotes the assertion $\bigwedge_{x \in \vartheta} x = \overleftarrow{x}$, $B \mid C$ denotes an assertion

characterising the 'relational composition' of B and C, in other words, $(s_1, s_2) \models B \mid C$ iff there is a state s_3 such that $(s_1, s_3) \models B$ and $(s_3, s_2) \models C$. Moreover, B^+ denotes an assertion characterising the transitive closure of B, B^* denotes an assertion characterising the reflexive and transitive closure of B, while A^B denotes an assertion characterising any state that can be reached from A by a finite number of B steps. This means that $s_1 \models A^B$ iff there is a state s_2 such that $(s_2, s_1) \models \overleftarrow{A} \wedge B^*$. Finally, wf B (where B is first-order) denotes an assertion which is valid iff B is well-founded on the set of states, in other words, iff there is no infinite sequence of states $s_1 s_2 \ \ldots \ s_k \ \ldots$ such that for all $j \geq 1$, $(s_j, s_{j+1}) \models B$.

The *consequence*-rule

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

is perhaps the easiest to understand. It basically states that it is always sound to strengthen the assumptions and weaken the commitments.

The *skip*-rule

$$\text{skip} \ \underline{\text{sat}} \ (\vartheta) :: (P, R, \text{false}, R^*)$$

is also rather trivial. Since any computation of a skip-statement (given the assumptions) is of the form

$$\langle \text{skip}, s_1 \rangle \xrightarrow{e} \ \ldots \ \xrightarrow{e} \langle \text{skip}, s_k \rangle \xrightarrow{i} \langle \varepsilon, s_k \rangle \xrightarrow{e} \ \ldots \ \xrightarrow{e} \langle \varepsilon, s_n \rangle,$$

where the only internal transition leaves the state unchanged, it is clear that false is the strongest possible guar-condition. Moreover, since any external transition, which changes the global state, is assumed to satisfy R, it follows that the overall effect is characterised by R^*.

The *assignment*-rule

$$\frac{\overleftarrow{P^R} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \langle v \rangle} \Rightarrow (G \vee I_\vartheta) \wedge E}{v := r \ \underline{\text{sat}} \ (\vartheta) :: (P, R, G, R^* \mid E \mid R^*)}$$

is more complicated. Any computation of an assignment-statement (given the assumptions) is of the form

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

Again, 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 that $s_k \models P^R$ and that $(s_k, s_{k+1}) \models \overleftarrow{P^R} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \langle v \rangle}$. But then it is clear from the premise that $(s_k, s_{k+1}) \models G \vee I_\vartheta$ and $(s_1, s_n) \models R^* \mid E \mid R^*$, which proves that the rule is sound.

The *block*-rule

$$\frac{z \ \underline{\text{sat}} \ (\vartheta) :: (P, R \wedge \bigwedge_{j=1}^{n} x_j = \overleftarrow{x_j}, G, E)}{\text{blo} \ x_1 : T_1, \ldots, x_n : T_n; z \ \text{olb} \ \underline{\text{sat}} \ (\vartheta \setminus \bigcup_{j=1}^{n} \{x_j\}) :: (P, R, G, E)}$$

can be used to 'hide' local variables. The rule is sound because the syntactic constraints on specified programs imply that $x_1, \ \ldots \ , x_n$ do not occur free in (P, R, G, E).

The *sequential*-rule

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

depends upon the fact that the first component's eff-condition implies the second component's pre-condition. This explains why P_2 occurs in the effect-condition of the first premise. Since an eff-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 $E_1 \mid E_2$.

With respect to the *if*-rule

$$\frac{z_1 \ \underline{\text{sat}}\ (\vartheta)::(P \wedge b, R, G, E) \qquad z_2 \ \underline{\text{sat}}\ (\vartheta)::(P \wedge \neg b, R, G, E)}{\text{if } b \text{ then } z_1 \text{ else } z_2 \text{ fi } \underline{\text{sat}}\ (\vartheta)::(P, R, G, E)}$$

it is important to remember that due to a syntactic constraint, the environment cannot access any variable occurring in b. Thus the truth value of the Boolean test cannot be changed by any process running in parallel. This means that for any computation

$$\langle z, s_1\rangle \xrightarrow{e} \langle z, s_2\rangle \xrightarrow{e} \ldots \xrightarrow{e} \langle z, s_k\rangle \xrightarrow{i} \sigma$$

of an if-statement z, there is a computation of the form

$$\langle z, s_1\rangle \xrightarrow{i} \langle z', s_1\rangle \xrightarrow{e} \langle z', s_2\rangle \ldots \xrightarrow{e} \langle z', s_k\rangle \xrightarrow{e} \sigma,$$

which can be obtained from the former by postponing the environment's interference until after the Boolean test has been evaluated. This means that for any computation of an if-statement if b then z_1 else z_2 fi in an environment characterised by P and R, there is a computation of z_1 in an environment characterised by $P \wedge b$ and R, or a computation of z_2 in an environment characterised by $P \wedge \neg b$ and R, with the same overall effect and with the same set of state changing internal transitions. Thus, it follows from the two premises that any internal transition which changes the global state satisfies G, and that the overall effect is characterised by E.

Because interference before the first evaluation of the Boolean test has no influence on the test's outcome, the soundness of the *while*-rule

$$\frac{\text{wf } F \qquad z \ \underline{\text{sat}}\ (\vartheta)::(P \wedge b, R, G, P \wedge F)}{\text{while } b \text{ do } z \text{ od } \underline{\text{sat}}\ (\vartheta)::(P, R, G, (F^+ \vee R^*) \wedge \neg b)}$$

follows by a similar argument. Observe that, for the same reason, the falsity of the Boolean test is preserved after the while-statement terminates. To prove termination, it is enough to show that an assertion characterising the effect of the loop's body in the actual environment is well-founded when considered as a binary relation on states. This explains the first premise. If the loop iterates at least once, then the overall effect is characterised by $F^+ \wedge \neg b$, while it is characterised by $R^* \wedge \neg b$ otherwise.

To grasp the intuition behind the *parallel*-rule

$$\frac{z_1 \ \underline{\text{sat}}\ (\vartheta)::(P, R \vee G_2, G_1, E_1) \qquad z_2 \ \underline{\text{sat}}\ (\vartheta)::(P, R \vee G_1, G_2, E_2)}{\{z_1 \parallel z_2\} \ \underline{\text{sat}}\ (\vartheta)::(P, R, G_1 \vee G_2, E_1 \wedge E_2)}$$

observe that the rely-condition of the first premise allows any interference due to z_2, and similarly that the rely-condition of the second premise allows any interference due to z_1. Thus since the eff-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 *domain*-rule

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

is straightforward. If the actual program is employed in a state which does not satisfy the pre-condition, then there are no constraints on its behaviour. Thus, the eff-condition can be restricted to transitions from states which satisfy the pre-condition.

The *access*-rule

$$\frac{z \text{ sat } (\vartheta)::(P,R \wedge x = \overleftarrow{x},G,E)}{z \text{ sat } (\vartheta)::(P,R,G,E)} \qquad \textit{where } x \in \textit{hid}[z] \cap \vartheta$$

is also needed. Under certain circumstances it allows the rely-condition to be weakened. The rule is sound because of the environment's restricted access to programming variables occurring in the Boolean test of an if- or a while-statement.

4 Synchronisation

4.1 Specified Programs

The last section explained how rely- and guar-conditions can be used to reason about interfering programs. Unfortunately, the rely/guarantee method cannot be employed for the development of programs whose correctness depends upon some sort of delay construct. The object of this section is to extend specifications with a fifth assertion, called a wait-condition, and show how this allows for synchronisation arguments.

A specification is now of the form $(\vartheta)::(P,R,W,G,E)$. The only new component is the wait-condition W, which is a unary assertion. The unhooked version of any free hooked or unhooked variable occurring in the assertions is still required to be an element of ϑ.

The assumptions about the environment are the same as before, but the commitments are changed. A configuration c is *blocked* if it is disabled and $Z(c) \neq \varepsilon$. Moreover, a computation *deadlocks* if it is finite and its final configuration is blocked. Given an environment which satisfies the assumptions, then the program is no longer required to terminate, but only to terminate or deadlock. The implementation is not allowed to deadlock 'inside' the body of an await-statement, and the final state is required to satisfy the wait-condition if the program deadlocks. Observe, that this implies that the program can only become blocked in a state which satisfies the wait-condition. On the other hand, the overall effect is constrained to satisfy the eff-condition if the program terminates.

Definition 6 *Given a glo-set ϑ, a wait-condition W, a guar-condition G, and an eff-condition E, then $int[\vartheta,W,G,E]$ denotes the set of all computations σ such that:*

- $len(\sigma) \neq \infty$,
- *for all* $1 \leq j < len(\sigma)$, *if* $L(\sigma_j) = i$ *and* $S(\sigma_j) \overset{\vartheta}{\neq} S(\sigma_{j+1})$ *then* $(S(\sigma_j), S(\sigma_{j+1})) \models G$,
- *if* $Z(\sigma_{len(\sigma)}) \neq \varepsilon$ *then* $S(\sigma_{len(\sigma)}) \models W$,
- *if* $Z(\sigma_{len(\sigma)}) = \varepsilon$ *then* $(S(\sigma_1), S(\sigma_{len(\sigma)})) \models E$.

Observe that if $ext[\vartheta,P,R] \cap cp[z]$ contains a computation which has at least one identity internal transition, it follows that $ext[\vartheta,P,R] \cap cp[z]$ has elements of infinite length. Thus, since the identity internal

transition is used to model a nonterminating await body, it follows from the first constraint that the bodies of await-statements are required to terminate.

(It is also possible to interpret the wait-condition as an assumption about the environment, in which case the actual program is assumed always eventually to be released given that it never becomes blocked in a state which does not satisfy the wait-condition. Definition 4 is in that case left unchanged, while definition 3 is extended with a fourth condition

- if $len(\sigma) \neq \infty$ and $Z(\sigma_{len(\sigma)}) \neq \varepsilon$, then $S(\sigma_{len(\sigma)}) \models \neg W$.

The deduction rules are exactly the same for both interpretations.)

Definition 7 *A specified program* z $\underline{\text{sat}}$ $(\vartheta)::(P,R,W,G,E)$ *is valid iff* $ext[\vartheta,P,R] \cap cp[z] \subseteq int[\vartheta,W,G,E]$.

This means that if z $\underline{\text{sat}}$ $(\vartheta,\alpha)::(P,R,W,G,E)$ is a valid specified program, z is executed in an environment characterised by P and R, and z is not infinitely overtaken, then z either deadlocks in a state which satisfies W or terminates in a state such that the overall effect is characterised by E. Thus, z is totally correct with respect to the same specification if W is equivalent to false.

4.2 Deduction Rules

With one exception, the *parallel*-rule, the changes to the rules given above are trivial. To grasp the intuition behind the new parallel-rule first consider

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

which is sufficiently strong whenever $\{z_1 \parallel z_2\}$ does not become blocked. It follows from the second premise that z_1 can only become blocked in a state which satisfies W_1 when executed in an environment characterised by P and $R \vee G_2$. Moreover, the third premise implies that z_2 can only become blocked in a state which satisfies 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 blocked after z_2 has terminated, that z_2 cannot be blocked after z_1 has terminated, and that z_1 and z_2 cannot be blocked at the same time, it follows that $\{z_1 \parallel z_2\}$ cannot become blocked in an environment characterised by P and R.

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

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

The idea is that W characterises the states in which the overall program may become blocked. This rule can of course be generalised further to deal with more than two processes:

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

Here, $\|_{j=1}^{m} z_j$ 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 is blocked in a state s such that $s \models \neg W \wedge W_j$, then there is at least one other process which is enabled. This rule is 'deducible' from the basic rules given above, and it will from now on be referred to as the *generalised* parallel-rule.

The *await*-rule

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

allows for synchronisation arguments. The statement can only become blocked in a state which does not satisfy the Boolean test b and can be reached from a state which satisfies the pre-condition P by a finite number of external transitions. This motivates the conclusion's wait-condition. The environment is constrained from interfering with the await-statement's body, which explains the choice of rely- and wait-conditions in the premise. Moreover, the await-statement's body is required to terminate for any state which satisfies $P^R \wedge b$. The rest should be obvious from the discussion above, since any computation of an await-statement has maximum of one internal transition which alters the global state.

5 Expressiveness

5.1 Motivation

It has been shown above, that interference can be dealt with by using rely- and guar-conditions, and that the introduction of a wait-condition allows for the design of programs whose correctness depends upon synchronisation. Unfortunately, many interesting developments are excluded because sufficiently strong intermediate assertions cannot be expressed. The object of this section is to show how the expressiveness can be increased by introducing auxiliary variables.

In [OG76] auxiliary variables are implemented as if they were ordinary programming variables, and then afterwards removed by a deduction rule specially designed for this purpose. This is not a very satisfactory method, because in some cases a large number of auxiliary variables are needed, and the procedure of first implementing them and then removing them is rather tedious. The approach in this paper is more in the style of [Sou84], where the auxiliary structure is only a part of the logic. Nevertheless, although it is possible to define history variables in LSP, auxiliary variables may be of any type, and it is up to the user to define the auxiliary structure he prefers. Auxiliary variables will be used for two different purposes:

- To strengthen a specification to eliminate undesirable implementations. In this case auxiliary variables are used as a *specification tool*; they are employed to characterise a program that has not yet been implemented.
- To strengthen a specification to make it possible to prove that a certain program satisfies a particular specification. Here auxiliary variables are used as a *verification tool*, since they are introduced to show that a given algorithm satisfies a specific property.

5.2 As a Specification Tool

An example, where auxiliary variables are used as a specification tool, will be discussed first. The task is to specify a program that adds a new element O to a global buffer called *buff*. If the environment is restricted from interfering with *buff*, then this can easily be expressed as follows:

$$(\{buff\}) :: (\mathsf{true}, \mathsf{false}, \mathsf{false}, buff = [O] \frown \overleftarrow{buff}, buff = [O] \frown \overleftarrow{buff})$$

($[O]$ denotes a sequence consisting of one element O, while $\frown$ is the usual concatenation operator on finite sequences.) The pre-condition states that an implementation must be applicable in any state. Moreover, the rely-condition restricts the environment from changing the value of *buff*, thus the eff-condition may be used to express the desired property. Finally, the guar-condition specifies that the concatenation step takes place in isolation, while the falsity of the wait-condition requires the implementation to terminate.

If the environment is allowed to interfere freely with *buff*, the task of formulating a specification becomes more difficult. Observe that the actual concatenation step is still required to be atomic; the only difference from above is that the environment may interfere immediately before and (or) after the concatenation takes place. Since there are no restrictions on the way the environment can change *buff*, and because external transitions, both before the first internal transition and after the last, are included in the overall effect, the eff-condition must allow anything to happen. This means that the eff-condition is no longer of much use. Thus, the guar-condition is the only hope to pin down the intended behaviour. The specification

$$(\{buff\}) : : (\text{true}, \text{true}, \text{false}, buff = [O] \frown \overleftarrow{buff}, \text{true})$$

is almost sufficient. The only problem is that there is no restriction on the number of times the implementation is allowed to add O to *buff*. Hence, skip is for example one possible implementation.

One solution is to introduce a Boolean auxiliary variable called *dn*, and use *dn* as a flag to indicate whether the implementation has added O to *buff* or not. To distinguish the auxiliary variables from the global programming variables, it has been found helpful to add a new component to a specification: an *aux-set* which contains the auxiliary variables. The program can then be specified as follows:

$$(\{buff\}, \{dn\}) : : (\neg dn, dn \Leftrightarrow \overleftarrow{dn}, \text{false}, buff = [O] \frown \overleftarrow{buff} \wedge \neg \overleftarrow{dn} \wedge dn, dn)$$

Since the environment cannot change the value of *dn*, the implementation only can add O to *buff* in a state where *dn* is false, the concatenation transition changes *dn* from false to true, and the implementation is not allowed to change *dn* from true to false, it follows from the pre- and eff-conditions that the implementation adds O to *buff* once and only once.

5.3 As a Verification Tool

The next example shows how auxiliary variables can be used as a verification tool. Without auxiliary structure,

$$(\{x\}) : : (\text{true}, x > \overleftarrow{x}, \text{false}, x = \overleftarrow{x} + 1 \vee x = \overleftarrow{x} + 2, x \geq \overleftarrow{x} + 3)$$

is the specification that gives the best possible characterisation of the program $x := x+1; x := x+2$, given the actual assumptions about the environment. Furthermore, this tuple is also the strongest possible specification for the program $x := x+2; x := x+1$ with respect to the same assumptions about the environment. Let z_1 denote the first program and z_2 the second. If the overall environment is constrained to leave x unchanged, it is clear that the parallel composition of z_1 and z_2 satisfies:

$$(\{x\}) : : (\text{true}, \text{false}, \text{false}, \text{true}, x = \overleftarrow{x} + 6).$$

Unfortunately, there is no way to deduce this only from the information in the specification of the components, because the component specification is, for example also satisfied by $x := x+2; x := x+2$.

The solution again is to introduce auxiliary variables. Let y_1 be a variable that records the overall effect of the updates to x in z_1, while y_2 records the overall effect of the updates to x in z_2. Clearly, it is required that z_1 and z_2 cannot change the value of y_2 respectively y_1. The specification of z_1 can then be rewritten as

$$\begin{aligned}(\{x\}, \{y_1, y_2\}) : : (&\text{true}, y_1 = \overleftarrow{y_1} \wedge x - \overleftarrow{x} = y_2 - \overleftarrow{y_2}, \text{false}, \\ &y_2 = \overleftarrow{y_2} \wedge x - \overleftarrow{x} = y_1 - \overleftarrow{y_1}, x = \overleftarrow{x} + 3 + (y_2 - \overleftarrow{y_2}) \wedge y_1 - \overleftarrow{y_1} = 3),\end{aligned}$$

while z_2 fulfills

$$(\{x\},\{y_1,y_2\})::(\text{true}, y_2 = \overleftarrow{y_2} \wedge x - \overleftarrow{x} = y_1 - \overleftarrow{y_1}, \text{false},$$
$$y_1 = \overleftarrow{y_1} \wedge x - \overleftarrow{x} = y_2 - \overleftarrow{y_2}, x = \overleftarrow{x} + (y_1 - \overleftarrow{y_1}) + 3 \wedge y_2 - \overleftarrow{y_2} = 3).$$

It follows easily from these two specifications that the overall effect of $\{z_1 \parallel z_2\}$ is characterised by $x = \overleftarrow{x} + 6$, if the overall environment is restricted from changing x.

5.4 Specified Programs

A specified program is now of the form $(\vartheta, \alpha)::(P, R, W, G, E)$. It is required that $\vartheta \cap \alpha = \{\}$, and that the unhooked version of any free hooked or unhooked variable occurring in the assertions is an element of $\vartheta \cup \alpha$.

To characterise what it means for a program to satisfy such a specification, it is necessary to introduce a new relation $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2$, called an *augmentation*, which states that the program z_2 can be obtained from the program z_1 by adding auxiliary structure constrained by the set of global programming variables ϑ and the set of auxiliary variables α. There are of course a number of restrictions on the auxiliary structure. First of all, to make sure that the auxiliary structure has no influence on the algorithm, auxiliary variables are constrained from occurring in the Boolean tests of if-, while- and await-statements. Furthermore, they cannot appear on the right-hand side of an assignment, unless the corresponding variable on the left-hand side is auxiliary. Moreover, since it must be possible to remove some auxiliary variables from a specified program without having to remove all the auxiliary variables, it is important that they do not depend upon each other. This means that if an auxiliary variable occurs on the left-hand side of an assignment-statement, then this is the only auxiliary variable that may occur in the corresponding expression on the right-hand side. In other words, to eliminate all occurrences of an auxiliary variable from a program, it is enough to remove all assignments with this variable on the left-hand side. However, an assignment to an auxiliary variable may have any number of elements of ϑ in its right-hand side expression. Finally, since auxiliary variables will only be employed to record information about state changes and synchronisation, auxiliary variables are only updated in connection with await- and assignment-statements.

Given two lists a, u of respectively variables and expressions, and two sets of variables ϑ, α, then $a \leftarrow_{(\vartheta,\alpha)} u$ denotes that a and u are of the same length, that any element of a is an element of α, and that any variable occurring in u's j'th expression is either an element of ϑ or equal to a's j'th variable. An augmentation can then be defined in a more formal way:

Definition 8 *Given two programs z_1, z_2 and two sets of variables ϑ and α, then $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2$, iff z_2 can be obtained from z_1 by substituting*

- *a statement of the form*

  $$v \frown a := r \frown u,$$

  where $a \leftarrow_{(\vartheta,\alpha)} u$, for each occurrence of an assignment-statement $v := r$, which does not occur in the body of an await-statement,

- *a statement of the form*

  await b do z'; $a := u$ od,

  where $z \overset{(\vartheta,\alpha)}{\hookrightarrow} z'$ and $a \leftarrow_{(\vartheta,\alpha)} u$, for each occurrence of an await-statement await b do z od, *which does not occur in the body of another await-statement.*

($v \frown a$ denotes a prefixed by v — similarly for $r \frown u$.) It is now straightforward to extend definition 7 to the general case:

Definition 9 *A specified program* z_1 sat $(\vartheta, \alpha)::(P,R,W,G,E)$ *is valid iff there is a program* z_2 *such that* $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2$ *and* $ext[\vartheta \cup \alpha, P, R] \cap cp[z_2] \subseteq int[\vartheta \cup \alpha, W, G, E]$.

5.5 Deduction Rules

With two exceptions, the assignment- and the await-rules, the deduction rules given above can be updated in an obvious way. With respect to the *assignment*-rule

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

remember that the execution of an assignment-statement $v := r$ actually corresponds to the execution of an assignment-statement of the form $v \frown a := r \frown u$. Thus, the only real difference from the above is that the premise must guarantee that the assignment-statement can be extended with auxiliary structure in such a way that the specified changes to both the auxiliary variables and the programming variables will indeed take place.

The *await*-rule

$$\frac{\begin{array}{l} E_1 \mid (I_\vartheta \wedge a = \overleftarrow{u} \wedge I_{\alpha\setminus(a)}) \Rightarrow (G \vee I_{\vartheta\cup\alpha}) \wedge E_2 \\ z \text{ sat } (\vartheta, \alpha)::(P^R \wedge b, \mathsf{false}, \mathsf{false}, \mathsf{true}, E_1) \end{array}}{\text{await } b \text{ do } z \text{ od sat } (\vartheta, \alpha)::(P,R,P^R \wedge \neg b, G, R^* \mid E_2 \mid R^*)} \qquad \textit{where } a \leftarrow_{(\vartheta,\alpha)} u$$

is closely related to the assignment-rule; there is only one state-changing internal transition, and auxiliary variables may be altered in the same atomic step. An extra premise is added to allow the auxiliary variables to be updated.

The *elimination*-rule

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

can be used to remove auxiliary structure from a specification. Remember, that due to the syntactic constraints on specifications it follows (from the fact that the conclusion is a specified program) that W, G and E have no occurrences of x.

6 Discussion

Some useful adaptation rules plus examples where LSP is used for the development of nontrivial algorithms can be found in [Stø90].

A soundness proof for LSP has been given in a rather informal way above. A more formal proof is included in [Stø90]. In [Stø90] it is also shown that LSP is relatively complete under the assumptions that structures are admissible, and that for any first order assertion A and structure π, it is always possible to express an assertion B in L, which is valid in π iff A is well-founded on the set of states in π.

Because the programming language is unfair, the system presented in this paper cannot deal with programs whose algorithms rely upon busy waiting. Thus LSP is incomplete with respect to a weakly fair language and even more so for a strongly fair programming language. However, this does not mean that fair languages cannot be dealt with in a similar style. In [Stø91] it is shown how LSP can be modified to handle both weakly fair and strongly fair programming languages. Only the while- and await-rules have to be changed.

The program constructs discussed in this paper are deterministic (although they all have a nondeterministic behaviour due to possible interference), and all functions have been required to be total. These constraints are not necessary. It is shown in [Stø90] that LSP can easily be extended to deal with both nondeterministic program constructs and partial functions.

This paper has only proposed a set of program-decomposition rules. How to formulate sufficiently strong data-refinement rules is still an open question. Jones [Jon81] proposed a refinement-rule for the rely/guarantee-method which can easily be extended to deal with LSP specifications. Unfortunately, as pointed out in [WD88], this refinement-rule is far from complete.

LSP can be thought of as a compositional reformulation of the Owicki/Gries method [OG76]. The rely-, guar- and wait-conditions have been introduced to avoid the final non-interference and freedom-from-deadlock proofs (their additional interference-freedom requirement for total correctness is not correct [AdBO90]). However, there are some additional differences. The programming language differs from theirs in several respects. First of all, variables occurring in the Boolean test of an if- or a while-statement are restricted from being accessed by the environment. In the Owicki/Gries language there is no such constraint. On the other hand, in their language await- and parallel-statements are constrained from occurring in the body of an await-statement. No such requirement is stated in this paper. The handling of auxiliary variables has also been changed. Auxiliary variables are only a part of the logic. Moreover, they can be employed both as a verification tool and as a specification tool, while in the Owicki/Gries method they can only be used as a verification tool.

Jones' system [Jon83] can be seen as a restricted version of LSP. There are two main differences. First of all, LSP has a wait-condition which makes it possible to deal with synchronisation. Secondly, because auxiliary variables may be employed both as specification and verification tools, LSP is more expressive.

Stirling's method [Sti88] employs a proof tuple closely related to that of Jones. The main difference is that the rely- and guar-conditions are represented as sets of invariants, while the post-condition is unary, not binary as in Jones' method. Auxiliary variables are implemented as if they were ordinary programming variables, and they cannot be used as a specification tool. Although this method favours top-down development in the style of Jones, it can only be employed for the design of partially correct programs.

Soundararajan [Sou84] uses CSP inspired history variables to state assumptions about the environment. Unfortunately, on many occasions, the use of history variables seems excessive. One advantage with LSP is therefore that the user is free to choose the auxiliary structure he prefers. Another difference is that LSP is not restricted to partial correctness.

Barringer, Kuiper and Pnueli [BKP84] employ temporal logic for the design of parallel programs. Their method can be used to develop nonterminating programs with respect to both safety and general liveness properties, and this formalism is therefore much more general than the one presented in this paper. However, although it is quite possible to employ the same temporal logic to develop totally correct sequential programs, most users would prefer to apply ordinary Hoare-logic in the style of for example VDM [Jon90]. The reason is that Hoare-logic is designed to deal with the sequential case only, and it is therefore both simpler to use and easier to understand than a formalism powerful enough to handle concurrency. A similar distinction can be made between the development of terminating programs versus programs that are not supposed to terminate and regarding different fairness constraints. LSP should be understood as a method specially designed for the development of totally correct shared-state parallel programs.

The Xu/He approach [XH91] is (as pointed out in their paper) inspired by LSP's tuple of five assertions. However, instead of a wait-condition they use a run-condition — the negation of LSP's wait. Another difference is their specification oriented semantics. Moreover, auxiliary variables are dealt with in the Owicki/Gries style. This means that auxiliary variables are implemented as if they were ordinary programming variables and cannot be used as a specification tool.

7 Acknowledgements

As mentioned above, this paper is based on the author's PhD-thesis, and I would first all like to thank my supervisor Cliff B. Jones for his help and support. I am also indebted to Xu Qiwen, Wojciech Penczek,

Howard Barringer and Mathai Joseph. Financial support has been received 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.

[AdBO90] K. R. Apt, F. S. de Boer, and E. R. Olderog. Proving termination of parallel programs. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty Is Our Business, A Birthday Salute to Edsger W. Dijkstra*. Springer-Verlag, 1990.

[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.

[FP78] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978.

[Jon81] C. B. Jones. *Development Methods for Computer Programs Including a Notion of Interference*. PhD thesis, Oxford University, 1981.

[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.

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

[Sou84] N. Soundararajan. A proof technique for parallel programs. *Theoretical Computer Science*, 31:13–29, 1984.

[Sti88] C. Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, University of Manchester, 1990.

[Stø91] K. Stølen. Proving total correctness with respect to fair (shared-state) parallel languages. In preparation, 1991.

[WD88] J. C. P. Woodcock and B. Dickinson. Using VDM with rely and guarantee-conditions. Experiences from a real project. In R. Bloomfield, L. Marshall, and R. Jones, editors, *Proc. 2nd VDM-Europe Symposium, Lecture Notes in Computer Science 328*, pages 434–458, 1988.

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