An Attempt to Reason about Shared-State Concurrency in the Style of VDM

Ketil Stølen[⋆]

Department of Computer Science, Manchester University,
Oxford Road, Manchester, M13, 9PL

Abstract. The paper presents an attempt to develop a totally correct shared-state parallel program in the style of VDM. Programs are specified by tuples of five assertions (P, R, W, G, E). The pre-condition P, the rely-condition R and the wait-condition W describe assumptions about the environment, while the guar-condition G and the eff-condition E characterise commitments to the implementation.

The pre-, rely- and guar-conditions are closely related to the similarly named conditions in Jones' rely/guarantee method, while the eff-condition corresponds to what Jones calls the post-condition. The wait-condition is supposed to characterise the set of states in which it is safe for the implementation to be blocked; in other words, the set of states in which the implementation, when it becomes blocked, eventually will be released by the environment. The implementation is not allowed to be blocked during the execution of an atomic statement.

Auxiliary variables are introduced to increase the expressiveness. They are used both as a specification tool; to characterise a program that has not yet been implemented, and as a verification tool; to show that a given algorithm satisfies a specific property. However, although it is possible to define history-variables in this approach, 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

Appeared in: Proc. VDM'91, LNCS 551, pages 324-342, Springer 1991.

VDM, the Vienna Development Method, has been used successfully for the development of software in a wide variety of areas (see for example [JS90]). However, VDM is basically a technique for the design of sequential programs. The object of this paper is to describe a method that can be used to reason about shared-state concurrency in a similar way.

The first attempt to develop shared-state parallel programs in a VDM-style was due to Cliff Jones [Jon81], [Jon83]. In his approach, often called the *rely/guarantee* method, a proof tuple is of the form

$$z \ \underline{\text{sat}}\ (P, R, G, Q),$$

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* about the *environment*. In return the *implementation* z must satisfy the guar(antee)-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 executing the implementation in such an environment.

[⋆] Address from September 1st 1991:Institut für Informatik, der Technischen Universität, postfach 20 24 20, Arcisstrasse 21, D-8000 München 2. email:stoelen@lan.informatik.tu-muenchen.de

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, the rely/guarantee method cannot be used for the development of algorithms whose correctness depends upon *synchronisation.* Furthermore, many valid developments are excluded because sufficiently strong intermediate assertions cannot be *expressed.* This paper, based on the authors PhD-thesis [Stø90], presents a method called LSP (Logic of Specified Programs), which can be thought of as an extension of the rely/guarantee approach, and which does not suffer from the two weaknesses pointed out above. (There are a number of minor changes with respect to [Stø90]; most notably, transitivity and reflexivity constraints have been removed.)

The paper is organised as follows: You are currently reading the introduction. Sections 2 and 3 give a brief overview of LSP. Some simplifying notation is introduced in section 4, while a non-trivial algorithm is developed in section 5. Finally, in section 6, some possible extensions are discussed, and LSP is compared with related approaches known from the literature.

2 Logic of Specified Programs

A *program* is a finite, nonempty list of symbols whose context-independent syntax is characterised in the well-known BNF-notation: Given that $\langle vl \rangle$, $\langle el \rangle$, $\langle dl \rangle$, $\langle ts \rangle$ 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

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

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, $\mathsf{blo}\ x{:}\,N, y{:}\,N; x, y{:=}\ 5+w, w\ \mathsf{olb}$ 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 updated by any process running in parallel. (If- and while-rules for a language without this requirement can be found in [Stø91b].) This constraint does of course not reduce the number of implementable algorithms. If x is a variable that can be updated by another process, then it is for example always possible to write $\mathsf{blo}\ y{:}\,N; y{:=}\ x; \mathsf{if}\ y = 0\ \mathsf{then}\ z_1\ \mathsf{else}\ z_2\ \mathsf{fi}\ \mathsf{olb}$ instead of $\mathsf{if}\ x =$

0 then z_1 else z_2 fi. Similar constraints are stated in [Sou84], [XH91]. For any program z, let $hid[z]$ denote the set of variables that occur in the Boolean test of an if- or a while-statement in z.

Since it is beyond the scope of this paper to give a soundness proof for LSP (a detailed proof can be found in [Stø90]), no formal semantics will be given. However, there are a few important requirements which must be pointed out. Assignment-statements and Boolean tests are atomic. The environment is restricted from interfering until the await-statement's body has terminated if an evaluation of its Boolean test comes out true, and the execution of the await-statement's body is modeled by one atomic step. A *state* is defined as a mapping of all programming variables to values. The expressions in the assignment-statement's expression-list are evaluated in the same state. The assignment of an empty list of expressions to an empty list of variables corresponds to the usual skip-statement and will be denoted by skip.

No *fairness* constraint is assumed. In other words, a process may be *infinitely overtaken* by another process. A program is *blocked* in a state, if the program has not terminated, and its subprocesses have either terminated or are waiting in front of an await-statement whose Boolean test is false (in the actual state).

Since the object of LSP is to prove total correctness, a *progress* property is needed; namely that a program will always progress unless it is infinitely overtaken by the environment, it is blocked, or it has *terminated*. Finally, to avoid unnecessary complications, all functions are required to be total.

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 expressions in L can be assigned meanings in the usual way. $\models A$ means that 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 a 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

$$(\vartheta, \alpha) :: (P, R, W, G, E),$$

where the *pre-condition* P, and the *wait-condition* W are unary assertions, and 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, while the *aux-set* α is the set of auxiliary variables. It is required that the unhooked version of any hooked or unhooked free variable occurring in P, R, W, G or E is an element of $\vartheta \cup \alpha$, and that $\vartheta \cap \alpha = \{\}$. The *global state* is the state restricted to $\vartheta \cup \alpha$.

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 atomic step by the environment, 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. 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.

Thirdly, it is assumed that if the implementation can be blocked only in states which satisfy the wait-condition, and can never be blocked inside the body of an await-statement, then the implementation will always eventually be released by the environment — in other words, under this condition, if the implementation becomes blocked, then the environment will eventually change the state in such a way that the implementation may progress. (The wait-condition can also be interpreted as a commitment to the implementation. See [Stø90], [Stø91a] for a detailed explanation.)

Finally, it is assumed that the environment can only perform a finite number of consecutive atomic steps. This means that the environment can only perform infinitely many atomic steps, if the implementation performs infinitely many atomic steps. 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]).

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 atomic step by the implementation, which changes the global state, is required to satisfy the guar-condition, and the overall effect of executing the implementation is constrained to satisfy the eff-condition. Observe, that interference both before the implementation's first atomic step and after the implementation's last atomic step is 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 auxiliary variables are employed to ensure the needed expressiveness. They are not first implemented and then afterwards removed by a specially designed deduction-rule as in the Owicki/Gries method [OG76]. Instead, the auxiliary variables are only a part of the logic. Moreover, they can be used in two different ways:

- 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 employed to show that a given algorithm satisfies a specific property.

The auxiliary variables may be of any type, and it is up to the user to define the auxiliary structure he prefers.

To characterise the use of auxiliary variables 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 in an expression on the right-hand side in an assignment-statement, 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 a occurs on the left-hand side of an assignment-statement, then the only auxiliary variable that may occur in the corresponding expression on the right-hand side is a. However, the right-hand side expression may have any number of occurrences of elements of ϑ. This means that to eliminate all occurrences of an auxiliary variable a from a program, it is enough to remove all assignments to a. Finally, it is necessary to update auxiliary variables only in connection with assignment- and await-statements.

Before giving a more formal definition, it is necessary to introduce some helpful notation. If l and k are finite lists, then $\langle l \rangle$ denotes the set of elements in l, while $l \frown k$ denotes the result of prefixing k with l. Finally, if a is a list of variables, u is a list of expressions, and ϑ and α are two sets of variables, then $a \leftarrow_{(\vartheta,\alpha)} u$ denotes that a and u have the same number of elements, that $\langle a \rangle \subseteq \alpha$, 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 (recursively) as follows:

- 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

    $$\mathsf{await}\ b\ \mathsf{do}\ z'; a{:}= u\ \mathsf{od},$$

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

A *specified program* is a pair of a program z and a specification ω, written $z\ \underline{\text{sat}}\ \omega$. It is required that for any variable x occurring in z, x is an element of ω's glo-set iff x is a global variable with respect to z. A specified program

$$z_1\ \underline{\text{sat}}\ (\vartheta, \alpha){::}(P, R, W, G, E)$$

is valid iff there is a program z_2, such that $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2$, and

- z_2 terminates,
- any atomic step by z_2, which changes the global state, satisfies the guar-condition G,
- the overall effect of executing z_2 satisfies the eff-condition E,

whenever the environment is such that

- the initial state satisfies the pre-condition P,
- any atomic step by the environment, which changes the global state, satisfies the rely-condition R,
- if z_2 cannot be blocked in a state which does not satisfy the wait-condition, and z_2 cannot be blocked inside the body of an await-statement, then z_2 will always eventually be released by the environment,
- z_2 is not infinitely overtaken by the environment.

As an example, consider the task of specifying a program which adds a constant A to a global buffer called Bf. If the environment is restricted from interfering with Bf, then this can easily be expressed as follows:

$$(\{Bf\},\{\})::(\mathbf{true},\mathbf{false},\mathbf{false},Bf=[A]\frown \overleftarrow{Bf},Bf=[A]\frown \overleftarrow{Bf})$$

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 Bf, which means that the eff-condition can be used to express the desired change of state. Finally, the guar-condition specifies that the concatenation step takes place as one atomic step, while the falsity of the wait-condition implies that a correct implementation cannot become blocked.

If the environment is allowed to interfere freely with Bf, then 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 Bf, and because interference due to the environment, both before the implementation's first atomic step, and after its last, is 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. The specification

$$(\{Bf\},\{\})::(\mathbf{true},\mathbf{true},\mathbf{false},Bf=[A]\frown \overleftarrow{Bf},\mathbf{true})$$

is almost sufficient. The only problem is that there is no restriction on the number of times the implementation is allowed to add A to Bf. Hence, skip is for example one correct implementation, while $Bf:=[A]\frown Bf;Bf:=[A]\frown Bf$ is another correct 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 A to Bf or not. The program can then be specified as follows:

$$(\{Bf\},\{Dn\})::(\neg Dn,Dn\Leftrightarrow \overleftarrow{Dn},\mathbf{false},Bf=[A]\frown \overleftarrow{Bf}\wedge\neg\overleftarrow{Dn}\wedge Dn,Dn).$$

Since the environment cannot change the value of Dn, the implementation can add A to Bf only 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 A to Bf once and only once.

So far there has been no real need for the wait-condition. However, if an implementation is restricted from adding A to Bf until the environment has switched on a Boolean flag Rd, it is necessary to use the wait-condition to express that a valid environment will eventually switch on Rd:

$$(\{Bf,Rd\},\{Dn\})::(\neg Dn,Dn\Leftrightarrow \overleftarrow{Dn},\neg Rd,Bf=[A]\frown \overleftarrow{Bf}\wedge\neg\overleftarrow{Dn}\wedge\overleftarrow{Rd}\wedge Dn,Dn)$$

The guar-condition implies that the implementation cannot change the value of Rd from false to true, and moreover that the implementation can add A to $Buff$ only when Rd is switched on.

3 Deduction Rules

The object of this section is to formulate a number of deduction-rules for the development of valid specified programs. Additional rules needed for the completeness proof, and some useful adaptation rules are given as an appendix.

Given a list of expressions r, a set of variables ϑ, and three assertions A, B and C, where at least A is unary, then $\overleftarrow{r}$ 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 $\bigwedge_{x\in\vartheta} x = \overleftarrow{x}$, while $B \mid C$ 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 $(s_2, s_1) \models \overleftarrow{A} \wedge B^*$.

The *consequence*-rule

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

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

The *assignment*-rule is formulated in two steps. The first version

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

is sufficient whenever the set of auxiliary variables is empty. Since the assignment-statement is atomic, there is only one atomic step due to the actual program. Moreover, the environment may interfere a finite number of times both before and after. Since the initial state is assumed to satisfy the pre-condition P, and any change of global state due to the environment is assumed to satisfy the rely-condition R, it follows that the atomic step in which the actual assignment takes place satisfies $\overleftarrow{P^R} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \langle v \rangle}$. But then it is clear from the premise that this atomic step satisfies G if the state is changed, and I_ϑ otherwise. Moreover, it follows that the overall effect is characterised by $R^* \mid E \mid R^*$.

To grasp the intuition behind the general rule

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

remember that the execution of an assignment-statement $v{:=}\, r$ actually corresponds to the execution of an assignment-statement of the form $v \frown a{:=}\, r \frown u$ where $a \leftarrow_{(\vartheta,\alpha)} u$. 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 *parallel*-rule is also easier to understand when designed in several steps. The first rule

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

is sufficient whenever neither of the two processes can become blocked. The important thing to realise is 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 implementation's first atomic step and after the implementation's last atomic step, it is clear from the two premises that $\{z_1 \parallel z_2\}$ terminates, that any atomic step by the implementation, 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

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

is sufficient whenever $\{z_1 \parallel z_2\}$ cannot become blocked. It follows from the second premise that z_1 can be blocked only 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 be blocked only 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.

This rule can easily be extended to deal with the general case:

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

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

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

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 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. The last rule is 'deducible' from the basic rules of LSP.

The *await*-rule is related to the assignment-rule. Again the explanation is split into two steps. The first version

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

is sufficient whenever the set of auxiliary variables is empty. The statement can be blocked only 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 R-steps. This motivates the conclusion's wait-condition. The environment is syntactically 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 follow easily from the discussion above.

With respect to the general version

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

remember that the execution of an await-statement await b do z od corresponds to the execution of a statement of the form await b do z'; $a := u$ od, where $z \overset{(\vartheta,\alpha)}{\hookrightarrow} z'$ and $a \leftarrow_{(\vartheta,\alpha)} u$. Thus, given the assumptions about the environment, the effect of z' is characterised by E_1, while E_2 characterises the effect of z'; $a := u$. Moreover, the atomic step representing the 'execution' of z'; $a := u$ satisfies G if the state is changed, and $I_{\vartheta \cup \alpha}$ otherwise. This explains the 'new' premise.

4 Simplifying Notation

To make specifications more readable, a *process* scheme

```
Name(In) Out
glo g_dcl
aux a_dcl

    pre  P
    rely R
    wait W
    guar G
    eff  E,
```

which corresponds to VDM's operation concept, has been found helpful. Not surprisingly, *Name* is the name of the process, *In* is the list of input parameters, while *Out* is the list of output parameters. Moreover, global variables are declared in *g_dcl*, while *a-dcl* is used to declare auxiliary variables. Finally, P, R, W, G and E denote respectively the pre-, rely-, wait-, guar- and eff-conditions. Input and output parameters should be interpreted in the same way as in [Jon90].

In VDM [Jon90] it is indicated in the declaration of a variable whether the operation has write access to the variable or only read access. If an operation has no write access to a variable, clearly its value will be left unchanged. An obvious generalisation to the concurrent case is to declare variables according to whether

– both the process and the environment have write access,

– the process has write access and the environment has only read access,

– the environment has write access and the process has only read access,

– both the process and the environment have only read access.

However, because of the existence of the await-statement, a process may update a global variable x in such a way that this is invisible from the outside, namely by ensuring that x always has the same value when an atomic statement terminates as x had when the same atomic statement was entered. Thus, it seems more sensible to use the following convention: For any variable x (global or auxiliary), then

– icec x (internal change, external change) — means that both the process and the environment can do observable changes to x,

– ices x (internal change, external stable) — means that only the process can do observable changes to x,

– isec x (internal stable, external change) — means that only the environment can do observable changes to x,

– ises x (internal stable, external stable) — means that neither the process nor the environment can do observable changes to x.

In the style of VDM, if it is clear from g_dcl or a_dcl that a particular variable cannot be changed (in an observable way) by a process, its environment or both, then this will not be restated in the rely-, guar- and eff-conditions of the process. For example, if isec x is a declaration in g_dcl, then it is clear that any atomic change of state by the process will satisfy $x = \overleftarrow{x}$, but to keep the specifications as simple as possible, this does not have to be restated in the guar-condition, although it may be added as an extra conjunct when proofs are undertaken.

When proving properties of programs it is often helpful to insert assertions into the code. For example, the sequential program

$$\{\mathbf{true}\}\ \mathsf{while}\ x > 0\ \mathsf{do}\ \{x > 0\}\ x := x - 1\ \mathsf{od}\ \{x \leq 0\}$$

has three such assertions. The first and last characterise respectively the initial and final states, while the one in the middle describes the state each time the program counter is 'situated' between the Boolean test and the assignment-statement. Programs will be annotated in a similar style below, but because the assertions may have occurrences of hooked variables, and because the environment may interfere, it is necessary to discuss the meaning of such assertions in more detail. Observe that annotated programs are not a part of the formal system LSP; annotated programs are introduced here only to make it easier for the reader to follow the argumentation.

The annotations will have occurrences of hooked variables when this is convenient. The hooked variables are supposed to refer to the initial state with respect to the particular piece of code in which the annotation occur. Moreover, the truth of the annotations is supposed to be maintained by the environment. For example, if the environment is restricted from updating x, then the annotated program

$$\{x = \overleftarrow{x}\}\ x := x + 5;\ \{x = \overleftarrow{x} + 5\}\ x := x + 3\ \{x = \overleftarrow{x} + 8\},$$

states that x equals its initial value until the first assignment-statement is executed, that the difference between the value of x and its initial value is 5, when the program counter is situated between the first and the second assignment-statement, and that the difference between the value of x and its initial value is 8 after the second assignment-statement has terminated. If the rely-condition is changed to $x > \overleftarrow{x}$, the annotations must be updated as below

$$\{x \geq \overleftarrow{x}\}\ x := x + 5;\ \{x \geq \overleftarrow{x} + 5\}\ x := x + 3\ \{x \geq \overleftarrow{x} + 8\}.$$

5 Set Partition

Given two non-empty, disjoint sets of natural numbers, S and L; the task is to design a program which terminates in a state where the maximum element of S is less than the minimum element of L. The sizes of the two sets must remain unchanged. Moreover, after termination, the union of S and L is required to equal the union of their initial values. It is assumed that the program will only be used in an environment which does not interfere with S and L.

This informal specification can be translated into a more mathematical notation:

$Sort()$
glo ices S: **set of N**, L: **set of N**
aux

pre $S \cap L = \{\,\} \wedge S \neq \{\,\} \wedge L \neq \{\,\}$
rely **false**
wait **false**
guar **true**
eff $\#S = \#\overleftarrow{S} \wedge \#L = \#\overleftarrow{L} \wedge$
$S \cup L = \overleftarrow{S} \cup \overleftarrow{L} \wedge max(S) < min(L).$

It follows from the declarations of S and L (and also from the rely-condition) that they will not be updated by the environment. The wait-condition implies that a correct implementation will never become blocked. The guar-condition allows the implementor to update S and L as he likes. The rest should be clear from the informal specification.

The algorithm (inspired from [Bar85], [Dij82]) employs two processes called respectively *Small* and *Large*. The basic idea is as follows:

– The process *Small* starts by finding the maximum element of S. This integer is sent on to *Large* and then subtracted from S. The task of *Large* is to add the received integer to L, and thereafter send the minimum element of L (which by then contains the integer just received from *Small*) back to *Small* and remove it from L. The process *Small* adds the element sent from *Large* to S. Then, if the maximum of S equals the integer just received from *Large*, it follows that the maximum of S is less than the minimum of L and the algorithm terminates. Otherwise, the whole procedure is repeated. Since the difference between the maximum of S and the minimum of L is decreased at each iteration, it follows that the program will eventually terminate.

The variables Mx: **N** and Mn: **N** simulate respectively 'the channel' from *Small* to *Large* and 'the channel' from *Large* to *Small*. To secure that the two processes stay in step, the Boolean variable Flg: **B** is introduced. When *Small* switches on *Flg*, it means that *Large* may read the next value from Mx, and when *Large* makes *Flg* false, it signals that Mn is ready to be read by *Small*. The adding, finding the maximum and sending section of *Small* is mutually exclusive with the adding, finding the minimum and sending section of *Large*. The only thing the process *Small* is allowed to do while *Flg* is true, is to remove from S the integer it just sent to *Large*. Similarly, when *Flg* is false, *Large* is only allowed to remove the element it just sent to *Small*.

This means that an implementation should be of the form

blo Mx: **N**, Mn: **N**, Flg: **B**; $Init()$; $Conc()$ olb

where *Init* initialises the local state, and *Conc* represents the parallel composition of *Small* and *Large*.

To make it easier to formulate and reason about properties satisfied by the concurrent part of the implementation, *Init* will simulate the first iteration of the algorithm; in other words, perform the first interchange of values. This means that:

$Init()$
glo ices S: **set of N**, L: **set of N**, Mx: **N**, Mn: **N**, Flg: **B**
aux

pre $S \cap L = \{\,\} \wedge S \neq \{\,\} \wedge L \neq \{\,\}$
rely **false**
wait **false**
guar **true**
eff $\#S = \#\overleftarrow{S} \wedge \#L = \#\overleftarrow{L} \wedge S \cup L = \overleftarrow{S} \cup \overleftarrow{L} \wedge$
$Mn \leq Mx \wedge S \cap L = \{\,\} \wedge Mx = max(S) \wedge$
$Mn \in S \wedge Mn < min(L) \wedge \neg Flg.$

Basically, *Init* simulates 'the sending' of one element in both directions. Thus, the next process to transfer a value (if necessary) is *Small*, which explains the restriction on *Flg*. Moreover, *Small* has already determined the 'new' maximum of S. The implementation of *Init* is not very challenging. The program below is obviously sufficient:

$Mx := max(S); L := L \cup \{Mx\}; S := S - \{Mx\};$
$Mn := min(L); S := S \cup \{Mn\}; L := L - \{Mn\};$
$Flg :=$ **false**; $Mx := max(S).$

The next step is to characterise a few properties that will be invariantly true for the concurrent part of the implementation. Since for both processes the previously sent element is removed before the actual process starts to look for a new integer to send, and since the processes stay in step, it follows that

$$S \cap L \subseteq \{Mn, Mx\}.$$

Moreover, because *Large* will return the integer just received if the maximum of S is less than the minimum of L, it is also true that

$$Mn \leq Mx.$$

To simplify the reasoning, let *uInv* (from now on called the *unary* invariant) denote the conjunction of these two assertions.

To ensure maintenance of the original integers it is required that any atomic change of state satisfies

$$S \cup L \cup \{Mx, Mn\} = \overleftarrow{S} \cup \overleftarrow{L} \cup \{\overleftarrow{Mx}, \overleftarrow{Mn}\}.$$

This is of course not enough on its own; however, if the conjunction of the eff-conditions of the two processes implies that $\{Mx, Mn\} \subseteq S \cup L$, it follows easily from the eff-condition of *Init* that the desired maintenance property is satisfied by the overall program.

Since the first interchange of elements has already taken place in *Init*, it is clear that any transition by either *Small* or *Large* will satisfy

$$Mx \leq \overleftarrow{Mx} \wedge Mn \geq \overleftarrow{Mn}.$$

The only possible change of state due to *Large* while *Flg* is false, is that Mn is removed from L, and the only possible change of state due to *Small* while *Flg* is true, is that Mx is removed from S. Moreover, since *Small* never updates Mn and L, and *Large* never updates Mx and S, it follows that any atomic step satisfies the two assertions

$$\neg\overleftarrow{\mathit{Flg}} \Rightarrow \mathit{Mn} = \overleftarrow{\mathit{Mn}} \wedge (L \cup \{\mathit{Mn}\} = \overleftarrow{L} \vee L = \overleftarrow{L}),$$
$$\overleftarrow{\mathit{Flg}} \Rightarrow \mathit{Mx} = \overleftarrow{\mathit{Mx}} \wedge (S \cup \{\mathit{Mx}\} = \overleftarrow{S} \vee S = \overleftarrow{S}).$$

To prove that the number of elements in S, when *Small* terminates, equals the set's initial size, and similarly for L, any atomic step should also satisfy the following two assertions:

$$\neg\overleftarrow{\mathit{Flg}} \wedge \mathit{Flg} \Rightarrow (\mathit{Mx} = \mathit{Mn} \vee \mathit{Mx} \notin L) \wedge \mathit{Mn} \in S,$$
$$\overleftarrow{\mathit{Flg}} \wedge \neg\mathit{Flg} \Rightarrow (\mathit{Mx} = \mathit{Mn} \vee \mathit{Mn} \notin S) \wedge \mathit{Mx} \in L.$$

Let *bInv* (from now on called the *binary* invariant) denote the conjunction of these six assertions.

It may be argued that *uInv* and *bInv* should have had arguments indicating which part of the global state they affect. However, this has been ignored here because of the large number of arguments needed. One way to reduce the number of arguments is to introduce records in the style of VDM.

The process *Small* will become blocked only when it is ready to enter its critical section and *Large* has not yet finished its critical section. This means that *Small* will wait only in a state which satisfies *Flg*. Similarly, it is clear that *Large* will be held back only in a state characterised by $\neg\mathit{Flg}$. The conjunction of these two assertions is obviously inconsistent, which means that *Small* and *Large* cannot be blocked at the same time. One way to make sure that neither of the processes can be blocked after the other process has terminated, is to introduce an auxiliary variable $\mathit{Trm}: \mathbf{B}$, which is required to be false initially and is first switched on when *Small* leaves its critical section for the last time; in other words, when *Mx* equals *Mn*. If the unary invariant *uInv* is strengthened with the conjunct

$$\mathit{Trm} \Rightarrow \mathit{Flg},$$

and the binary invariant *bInv* is strengthened with the two conjuncts

$$\neg\overleftarrow{\mathit{Trm}} \wedge \mathit{Trm} \Leftrightarrow \neg\overleftarrow{\mathit{Flg}} \wedge \mathit{Flg} \wedge \mathit{Mx} = \mathit{Mn},$$
$$\overleftarrow{\mathit{Trm}} \Rightarrow \mathit{Trm}.$$

it follows that *Trm* is true after *Large* has terminated, and that *Flg* is true after *Small* has terminated. But then, since *Small* can become blocked only in a state which satisfies $\mathit{Flg} \wedge \neg\mathit{Trm}$, and *Large* can become blocked only in a state which satisfies $\neg\mathit{Flg}$, it follows that deadlock is impossible.

From the discussion above it is clear that *Small* does not need write access to L and *Mn*. Similarly, *Large* will never have to change the values of S, *Mx* and *Trm*. To secure mutual exclusion *Large* must maintain the falsity of *Flg*, while *Small* in return must guarantee never to make *Flg* false. Thus, in a more formal notation:

```
$Small()$
glo ices $S$: $\mathbf{set\ of\ N}$, $Mx$: $\mathbf{N}$
    isec $L$: $\mathbf{set\ of\ N}$, $Mn$: $\mathbf{N}$
    icec $Flg$: $\mathbf{B}$
aux ices $Trm$: $\mathbf{B}$

  pre  $Mx = max(S) \wedge Mx \notin L \wedge$
       $Mn \in S \wedge \neg Flg \wedge \neg Trm \wedge uInv$
  rely $(\neg \overleftarrow{Flg} \Rightarrow \neg Flg) \wedge bInv \wedge uInv$
  wait $Flg \wedge \neg Trm$
  guar $(\overleftarrow{Flg} \Rightarrow Flg) \wedge bInv \wedge uInv$
  eff  $\#S = \#\overleftarrow{S} \wedge Mx = Mn \wedge$
       $Mx = max(S) \wedge Flg$,
```

```
$Large()$
glo isec $S$: $\mathbf{set\ of\ N}$, $Mx$: $\mathbf{N}$
    ices $L$: $\mathbf{set\ of\ N}$, $Mn$: $\mathbf{N}$
    icec $Flg$: $\mathbf{B}$
aux isec $Trm$: $\mathbf{B}$

  pre  $Mx \notin L \wedge Mn \in S \wedge$
       $Mn < min(L) \wedge \neg Flg \wedge \neg Trm \wedge uInv$
  rely $(\overleftarrow{Flg} \Rightarrow Flg) \wedge bInv \wedge uInv$
  wait $\neg Flg$
  guar $(\neg \overleftarrow{Flg} \Rightarrow \neg Flg) \wedge bInv \wedge uInv$
  eff  $\#L = \#\overleftarrow{L} \wedge Mx = Mn \wedge$
       $Mn < min(L) \wedge Trm$.
```

Since the wait-conditions of both processes are inconsistent with the wait- and eff-conditions of the other process, it follows by the eff- (see appendix), consequence-, and parallel-rules that the concurrent part of the implementation satisfies:

```
$Conc()$
glo ices $S$: $\mathbf{set\ of\ N}$, $L$: $\mathbf{set\ of\ N}$, $Mx$: $\mathbf{N}$, $Mn$: $\mathbf{N}$, $Flg$: $\mathbf{B}$
aux ices $Trm$: $\mathbf{B}$

  pre  $Mx = max(S) \wedge Mx \notin L \wedge Mn \in S \wedge Mn < min(L) \wedge \neg Flg \wedge \neg Trm \wedge uInv$
  rely $\mathbf{false}$
  wait $\mathbf{false}$
  guar $\mathbf{true}$
  eff  $\#S = \#\overleftarrow{S} \wedge \#L = \#\overleftarrow{L} \wedge Mx = Mn \wedge Mx = max(S) \wedge Mn < min(L) \wedge bInv^* \wedge uInv$,
```

which together with *Init* gives the desired overall effect.

How should *Small* and *Large* best be decomposed? Obviously, in both cases a while-construct is needed. One possible strategy is the following:

```
blo $V_S$: $\mathbf{B}$;
    $V_S := (Mx \neq Mn)$;
    while $V_S$ do
        $Sml()$; $V_S := (Mx \neq Mn)$
    od;
    $Flg := \mathbf{true}$
olb,
```

```
blo $V_L$: $\mathbf{B}$;
    await $Flg$ do skip od;
    $V_L := (Mx \neq Mn)$;
    while $V_L$ do
        $Lrg()$; $V_L := (Mx \neq Mn)$
    od
olb.
```

For both loops the obvious termination expression is $Mx - Mn$. Moreover, since

$$\overleftarrow{Mn} < \overleftarrow{Mx} \wedge (Mx < \overleftarrow{Mx} \vee Mx = Mn) \wedge bInv \wedge uInv \Rightarrow 0 \leq Mx - Mn < \overleftarrow{Mx} - \overleftarrow{Mn},$$
$$\overleftarrow{Mn} < \overleftarrow{Mx} \wedge Mn > \overleftarrow{Mn} \wedge bInv \wedge uInv \Rightarrow 0 \leq Mx - Mn < \overleftarrow{Mx} - \overleftarrow{Mn},$$

it follows easily that both loops terminate, and that the specifications of *Small* and *Large* are satisfied, if it can be proved that *Sml* and *Lrg* are characterised by:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{Sml}()} \\
\mathsf{glo} & \text{ices } S\colon \mathbf{set\ of\ N}, \mathit{Mx}\colon \mathbf{N} \\
 & \text{isec } L\colon \mathbf{set\ of\ N}, \mathit{Mn}\colon \mathbf{N} \\
 & \text{icec } \mathit{Flg}\colon \mathbf{B} \\
\mathsf{aux} & \text{ices } \mathit{Trm}\colon \mathbf{B} \\
\\
\mathsf{pre} & \mathit{Mn} < \mathit{Mx} \wedge \mathit{Mx} = \mathit{max}(S) \wedge \mathit{Mx} \notin L \wedge \\
 & \mathit{Mn} \in S \wedge \neg \mathit{Flg} \wedge \neg\, \mathit{Trm} \wedge \mathit{uInv} \\
\mathsf{rely} & (\neg \overleftarrow{\mathit{Flg}} \Rightarrow \neg \mathit{Flg}) \wedge \mathit{bInv} \wedge \mathit{uInv} \\
\mathsf{wait} & \mathit{Flg} \wedge \neg\, \mathit{Trm} \\
\mathsf{guar} & (\overleftarrow{\mathit{Flg}} \Rightarrow \mathit{Flg}) \wedge \mathit{bInv} \wedge \mathit{uInv} \\
\mathsf{eff} & \#S = \#\overleftarrow{S} \wedge \mathit{Mx} = \mathit{max}(S) \wedge \mathit{Mn} \in S \wedge \\
 & (\mathit{Mx} < \overleftarrow{\mathit{Mx}} \vee \mathit{Mx} = \mathit{Mn}) \wedge \neg \mathit{Flg} \wedge \neg\, \mathit{Trm},
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{Lrg}()} \\
\mathsf{glo} & \text{isec } S\colon \mathbf{set\ of\ N}, \mathit{Mx}\colon \mathbf{N} \\
 & \text{ices } L\colon \mathbf{set\ of\ N}, \mathit{Mn}\colon \mathbf{N} \\
 & \text{icec } \mathit{Flg}\colon \mathbf{B} \\
\mathsf{aux} & \text{isec } \mathit{Trm}\colon \mathbf{B} \\
\\
\mathsf{pre} & \mathit{Mn} < \mathit{Mx} \wedge \mathit{Mx} \notin L \wedge \mathit{Mn} \in S \wedge \\
 & \mathit{Mn} < \mathit{min}(L) \wedge \mathit{Flg} \wedge \neg\, \mathit{Trm} \wedge \mathit{uInv} \\
\mathsf{rely} & (\overleftarrow{\mathit{Flg}} \Rightarrow \mathit{Flg}) \wedge \mathit{bInv} \wedge \mathit{uInv} \\
\mathsf{wait} & \neg \mathit{Flg} \\
\mathsf{guar} & (\neg \overleftarrow{\mathit{Flg}} \Rightarrow \neg \mathit{Flg}) \wedge \mathit{bInv} \wedge \mathit{uInv} \\
\mathsf{eff} & \#L = \#\overleftarrow{L} \wedge \mathit{Mn} < \mathit{min}(L) \wedge \\
 & \mathit{Mn} > \overleftarrow{\mathit{Mn}} \wedge (\mathit{Mx} = \mathit{Mn} \Leftrightarrow \mathit{Trm}).
\end{array}$$

It can be shown that *Sml* is satisfied by the following annotated program:

$$\{\#S = \#\overleftarrow{S} \wedge \mathit{Mx} = \overleftarrow{\mathit{Mx}} \wedge \mathit{Mn} < \mathit{Mx} \wedge \mathit{Mx} = \mathit{max}(S) \wedge \mathit{Mn} \in S \wedge \mathit{Mx} \notin L \wedge \\ \neg \mathit{Flg} \wedge \neg\, \mathit{Trm} \wedge \mathit{bInv}^* \wedge \mathit{uInv}\}$$

$\mathit{Flg} := \mathbf{true}$;

$$\{\#S = \#\overleftarrow{S} \wedge \mathit{Mx} = \overleftarrow{\mathit{Mx}} \wedge \mathit{Mx} = \mathit{max}(S) \wedge \\ (\neg \mathit{Flg} \Rightarrow \mathit{Mx} = \mathit{Mn} \vee (\mathit{Mn} \notin S \wedge \mathit{Mx} \in L)) \wedge \neg\, \mathit{Trm} \wedge \mathit{bInv}^* \wedge \mathit{uInv}\}$$

$S := S \setminus \{\mathit{Mx}\}$;

$$\{\#S = \#\overleftarrow{S} - 1 \wedge \mathit{Mx} = \overleftarrow{\mathit{Mx}} \wedge (S \neq \{\,\} \Rightarrow \mathit{Mx} > \mathit{max}(S)) \wedge \\ (\neg \mathit{Flg} \Rightarrow \mathit{Mx} = \mathit{Mn} \vee (\mathit{Mn} \notin S \wedge \mathit{Mx} \in L)) \wedge \neg\, \mathit{Trm} \wedge \mathit{bInv}^* \wedge \mathit{uInv}\}$$

await $\neg \mathit{Flg}$ do skip od;

$$\{\#S = \#\overleftarrow{S} - 1 \wedge \mathit{Mx} = \overleftarrow{\mathit{Mx}} \wedge (S \neq \{\,\} \Rightarrow \mathit{Mx} > \mathit{max}(S)) \wedge \\ (\mathit{Mx} = \mathit{Mn} \vee (\mathit{Mn} \notin S \wedge \mathit{Mx} \in L)) \wedge \neg \mathit{Flg} \wedge \neg\, \mathit{Trm} \wedge \mathit{bInv}^* \wedge \mathit{uInv}\}$$

$S := S \cup \{\mathit{Mn}\}$;

$$\{\#S = \#\overleftarrow{S} \wedge \mathit{Mx} = \overleftarrow{\mathit{Mx}} \wedge ((\mathit{Mx} > \mathit{max}(S) \wedge \mathit{Mx} \in L) \vee (\mathit{Mx} = \mathit{max}(S) \wedge \mathit{Mx} = \mathit{Mn})) \wedge \\ \mathit{Mn} \in S \wedge \neg \mathit{Flg} \wedge \neg\, \mathit{Trm} \wedge \mathit{bInv}^* \wedge \mathit{uInv}\}$$

$\mathit{Mx} := \mathit{max}(S)$;

$$\{\#S = \#\overleftarrow{S} \wedge ((\mathit{Mx} < \overleftarrow{\mathit{Mx}} \wedge \mathit{Mx} \notin L) \vee \mathit{Mx} = \mathit{Mn}) \wedge \mathit{Mx} = \mathit{max}(S) \wedge \\ \mathit{Mn} \in S \wedge \neg \mathit{Flg} \wedge \neg\, \mathit{Trm} \wedge \mathit{bInv}^* \wedge \mathit{uInv}\}.$$

Similarly, *Lrg* can be implemented as below:

$$\{\#L = \#\overleftarrow{L} \wedge Mn = \overleftarrow{Mn} \wedge Mn < Mx \wedge Mx \notin L \wedge Mn \in S \wedge$$
$$Mn < min(L) \wedge Flg \wedge \neg Trm \wedge bInv^* \wedge uInv\}$$

$L\!:= L \cup \{Mx\};$

$$\{\#L = \#\overleftarrow{L} + 1 \wedge Mn = \overleftarrow{Mn} \wedge Mx \in L \wedge Mn \in S \wedge$$
$$Mn < min(L) \wedge Flg \wedge \neg Trm \wedge bInv^* \wedge uInv\}$$

$Mn\!:= min(L);$

$$\{\#L = \#\overleftarrow{L} + 1 \wedge Mn > \overleftarrow{Mn} \wedge Mx \in L \wedge (Mx = Mn \vee Mn \notin S) \wedge$$
$$Mn = min(L) \wedge Flg \wedge \neg Trm \wedge bInv^* \wedge uInv\}$$

$Flg\!:=$ **false**;

$$\{\#L = \#\overleftarrow{L} + 1 \wedge Mn > \overleftarrow{Mn} \wedge (Flg \Rightarrow Mx = Mn \vee (Mx \notin L \wedge Mn \in S)) \wedge$$
$$Mn = min(L) \wedge (Flg \Rightarrow (Mx = Mn \Leftrightarrow Trm)) \wedge bInv^* \wedge uInv\}$$

$L\!:= L \setminus \{Mn\};$

$$\{\#L = \#\overleftarrow{L} \wedge Mn > \overleftarrow{Mn} \wedge (Flg \Rightarrow Mx = Mn \vee (Mx \notin L \wedge Mn \in S)) \wedge$$
$$Mn < min(L) \wedge (Flg \Rightarrow (Mx = Mn \Leftrightarrow Trm)) \wedge bInv^* \wedge uInv\}$$

await Flg do skip od

$$\{\#L = \#\overleftarrow{L} \wedge Mn > \overleftarrow{Mn} \wedge (Mx = Mn \vee (Mx \notin L \wedge Mn \in S)) \wedge$$
$$Mn < min(L) \wedge Flg \wedge (Mx = Mn \Leftrightarrow Trm) \wedge bInv^* \wedge uInv\}.$$

6 Discussion

It has been shown above how LSP can be employed to reason about shared-state concurrency in the style of VDM. The use of two invariants, a unary invariant, which is true initially and maintained by each atomic step, and a binary invariant, which is satisfied by any atomic change to the global state, simplified the design of the set-partition algorithm. This way of structuring a development has also been used on other examples with similar effect [Stø90]. Related invariant concepts are discussed in [Jon81], [GR89], [XH91].

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.

In [Stø90] LSP is proved to be sound with respect to an operational semantics, and 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 [Stø91b] it is shown how LSP can be modified to handle both weakly fair and strongly fair programming languages.

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 be extended to facilitate both nondeterministic program constructs and partial functions.

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. 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] — namely that a program's specification always can be verified on the basis of the specifications of its constitute components, without knowledge of the interior program structure of those components.

LSP can be thought of as a compositional reformulation of the Owicki/Gries method. 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 updated 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.

In LSP, and in most of the methods mentioned above, the syntactic structure of the programming language is used to direct the decomposition of a specification into subspecifications. Some argue that the syntactic structure of a programming language is too close to machine architecture and therefore less suited to guide the design of algorithms — at least at the most abstract levels. Ideas like this have lead to the proposal of action based formalisms like [Bac88], [CM88], [Lam90].

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 Howard Barringer, John S. Fitzgerald, Mathai Joseph, Wojciech Penczek and Xu Qiwen. Financial support has been received from the Norwegian Research Council for Science and the Humanities and the Wolfson Foundation.

References

[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.

[Bac88] R. J. R. Back. A calculus of refinments for program derivations. *Acta Informatica*, 25:593–624, 1988.

[Bar85] H. Barringer. *A Survey of Verification Techniques for Parallel Programs*, volume 191 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[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.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison-Wesley, 1988.

[Dij82] E. W. Dijkstra. A correctness proof for communicating processes: A small exercise. In *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, 1982.

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

[GR89] D. Grosvenor and A. Robinson. An evaluation of rely-guarantee. Unpublished Paper, March 1989.

[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. North-Holland, 1983.

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

[JS90] C. B. Jones and R. C. F. Shaw, editors. *Case Studies in Systematic Software Development.* Prentice-Hall International, 1990.

[Lam90] L. Lamport. A temporal logic of actions. Technical Report 57, Digital, Palo Alto, 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.

[Stø91a] K. Stølen. A method for the development of totally correct shared-state parallel programs. Accepted for CONCUR'91, proceedings will appear in Lecture Notes in Computer Science, 1991.

[Stø91b] 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.

Additional Rules Needed in Completeness Proof

if: :

$$\frac{\begin{array}{l} z_1\ \underline{\mathsf{sat}}\ (\vartheta,\alpha){:}{:}\,(P \wedge b, R, W, G, E) \\ z_2\ \underline{\mathsf{sat}}\ (\vartheta,\alpha){:}{:}\,(P \wedge \neg b, R, W, G, E) \end{array}}{\mathsf{if}\ b\ \mathsf{then}\ z_1\ \mathsf{else}\ z_2\ \mathsf{fi}\ \underline{\mathsf{sat}}\ (\vartheta,\alpha){:}{:}\,(P, R, W, G, E)}$$

pre: :

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

while: :

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

access: :

$$\frac{\begin{array}{l} x \in \textit{hid}[z] \\ z\ \underline{\mathsf{sat}}\ (\vartheta,\alpha){:}{:}\,(P, R \wedge x = \overleftarrow{x}, W, G, E) \end{array}}{z\ \underline{\mathsf{sat}}\ (\vartheta,\alpha){:}{:}\,(P, R, W, G, E)}$$

sequential: :

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

elimination: :

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

block: :

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

Some Useful Adaptation Rules

eff: :

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

rely: :

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

augment: :

$$\frac{\begin{array}{l} z_2 \overset{(\vartheta \setminus \alpha, \alpha)}{\hookrightarrow} z_1 \\ z_1\ \underline{\mathsf{sat}}\ (\vartheta, \{\,\}){:}{:}\,(P, R, W, G, E) \end{array}}{z_2\ \underline{\mathsf{sat}}\ (\vartheta \setminus \alpha, \alpha){:}{:}\,(P, R, W, G, E)}$$

invariant: :

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

stutter: :

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

glo::

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

aux::

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